May 1994

Querying Nested Collections

Limsoon Wong
University of Pennsylvania

University of Pennsylvania Institute for Research in Cognitive Science Technical Report No. IRCS-94-09.

Querying Nested Collections

Abstract
This dissertation investigates a new approach to query languages inspired by structural recursion and by the categorical notion of a monad.

A language based on these principles has been designed and studied. It is found to have the strength of several widely known relational languages but without their weaknesses. This language and its various extensions are shown to exhibit a conservative extension property, which indicates that the depth of nesting of collections in intermediate data has no effect on their expressive power. These languages also exhibit the finite-cofiniteness property on many classes of queries. These two properties provide easy answers to several hitherto unresolved conjectures on query languages that are more realistic than the flat relational algebra.

A useful rewrite system has been derived from the equational theory of monads. It forms the core of a source-to-source optimizer capable of performing filter promotion, code motion, and loop fusion. Scanning routines and printing routines are considered as part of optimization process. An operational semantics that is a blending of eager evaluation and lazy evaluation is suggested in conjunction with these input-output routines. This strategy leads to a reduction in space consumption and a faster response time while preserving good total time performance. Additional optimization rules have been systematically introduced to cache and index small relations, to map monad operations to several classical join operators, to cache large intermediate relations, and to push monad operations to external servers.

A query system Kleisli and a high-level query language CPL for it have been built on top of the functional language ML. Many of my theoretical and practical contributions have been physically realized in Kleisli and CPL. In addition, I have explored the idea of open system in my implementation. Dynamic extension of the system with new primitives, cost functions, optimization rules, scanners, and writers are fully supported. As a consequence, my system can be easily connected to external data sources. In particular, it has been successfully applied to integrate several genetic data sources which include relational databases, structured files, as well as data generated by special application programs.

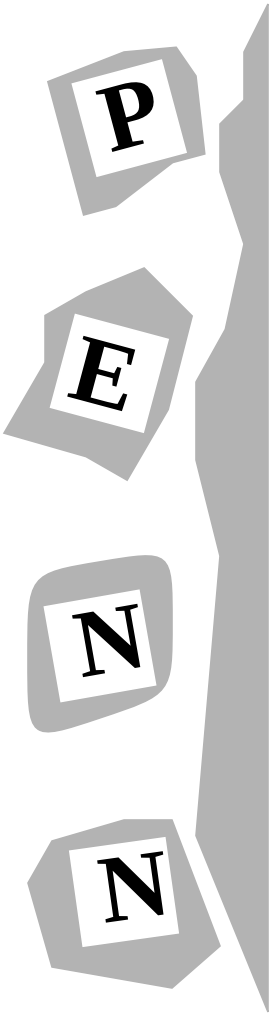
Comments
University of Pennsylvania Institute for Research in Cognitive Science Technical Report No. IRCS-94-09.

The Institute For Research In Cognitive Science

Querying Nested Collections (Ph.D. Dissertation)

by

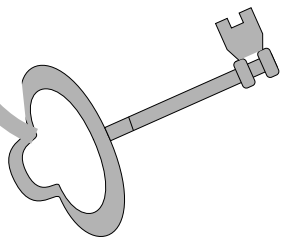
Limsoon Wong



University of Pennsylvania
3401 Walnut Street, Suite 400C
Philadelphia, PA 19104-6228

May 1994

Site of the NSF Science and Technology Center for Research in Cognitive Science



University of Pennsylvania
Founded by Benjamin Franklin in 1740

IRCS Report 94-09

QUERYING NESTED COLLECTIONS

LIMSOON WONG

A DISSERTATION

IN

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1994

Peter Buneman

Supervisor of Dissertation

Mark Steedman

Graduate Group Chairman

For my parents

ACKNOWLEDGEMENTS

My work in this dissertation is part of a greater project at the University of Pennsylvania to understand collection types and their query languages. I have benefited greatly from the weekly discussion at the "Tuesday Club," whose regular members are Peter Buneman, Susan Davidson, Wenfei Fan, Anthony Kosky, Leonid Libkin, Rona Machlin, Dan Suciu, Val Tannen, and myself. I have always looked forward to the intellectual and not-so-intellectual entertainments at our meetings.

I could not have begun my research on query languages without the guidance of Peter Buneman and Val Tannen. I could not have explored as much theoretical territory without the help of Peter Buneman, Dan Suciu, Val Tannen, and especially Leonid Libkin. I would not have touched query optimization without the push from Susan Davidson, David Maier, and Jonathan Smith. I would not have considered applying my system to the biomedical area without the vision of Peter Buneman, Susan Davidson, and Chris Overton; and I could not have succeeded without the help of Kyle Hart. I am grateful to them for the encouragement and help they have given me.

I have received useful suggestions, intriguing questions, and kind encouragement from Catriel Beeri, Peter Buneman, Jan Van den Bussche, John W. Carr III, Edward P. F. Chan, Susan Davidson, Guozhu Dong, Ron Fagin, Leonidas Fegaras, Stephane Grumbach, Dirk Van Gucht, Kyle Hart, Rick Hull, Dominic Juhasz, Paris Kanellakis, Anthony Kosky, Leonid Libkin, Tok Wang Ling, David Maier, Florian Matthes, Tova Milo, Teow Hin Ngair, Shamim Naqvi, Chris Overton, Jan Paredaens, Jong Park, Yatin Saraiya, Andre Scedrov, Joachim Schmidt, Michael Siff, Jonathan Smith, Jianwen Su, Dan Suciu, Val Tannen, Yong Chiang Tay, Philip Trinder, Moshe Vardi, Victor Vianu, Nghi Vo, and Scott Weinstein. I thank them for the discussions we had. I thank also Florian Matthes and Joachim Schmidt for hosting me when I visited Hamburg.

In the last three years, my research work has covered many topics on querying nested collections. It was hard for me to decide what material to include in my dissertation. My supervisor and my committee greatly influenced the selection in this dissertation. Peter

Buneman is my supervisor and Susan Davidson, Jean Gallier, David Maier, Jonathan Smith, Val Tannen, and Scott Weinstein served on my dissertation committee. I thank them for guiding me in preparing a dissertation that has a better balance between theory, practice, and application. Peter Buneman and Susan Davidson received some abuse from me when I was disgruntled by the committee's reaction when I proposed to include only the "purest" theories in my dissertation. I am grateful to them for being so tolerant and supportive.

I first discovered the logician trait in me at the Imperial College of Science, Technology, and Medicine. The lectures of Samson Abramsky, Krysia Broda, and Mike Smyth were largely responsible for it. This trait was considerably enhanced after my arrival at the University of Pennsylvania. The lectures of Carl Gunter, Andre Scedrov, Val Tannen, and Scott Weinstein were largely responsible for it. I am grateful to them for showing me the beauty, the breadth, and the depth of logic in computation.

I could write computer programs since I was a boy. But it was at the Imperial College of Science, Technology, and Medicine that I learned the art of programming. My love of functional programming can be attributed to the elegant programs I saw in the classes of Roger Bailey and to the beautiful transformations I saw in the classes of John Darlington. My partiality to open extensible systems can be attributed to the software engineering classes of Jeff Kramer and to the graphic systems classes of Duncan Gillies. I hope one day I will achieve their clarity in building systems large and small.

When I first looked for a job, Tom Maibaum and Martin Sadler promoted me at the Hewlett Packard Laboratories. When I first looked for a graduate school, Tom Maibaum and Mike Smyth promoted me at the University of Pennsylvania. When I was near completing my dissertation, Martin Sadler again offered to sing my praises at the Hewlett Packard Laboratories; I hope one day ambition will allow me to take up his offer. I am grateful to and am touched by their friendship and support.

My research productivity has been increased by the "extracurricular" activities provided by fellow Pennsylvanians Peter Buneman, Susan Davidson, Carl Gunter, Myra VanInwegen, Chuck Liang, Leonid Libkin, Teow Hin Ngair (now at the Institute of Systems Science in

Singapore), Jinah Park, Jong Park, Dan Suciu, Ivan Tam, Val Tannen, and Jian Zhang. I enjoy going to their homes and to restaurants, movies, zoos, parks, and other places with them.

Not to be forgotten are the invaluable help provided by members of Penn's research computing staff Mark Foster and Mark-Jason Dominus; administrative staff Elaine Benedetto, Nan Biltz, Jackie Caliman, Karen Carter, Michael Felker, and Billie Holland; and business office staff Gail Shannon. I thank fellow ex-members of the office committee Eric Brill and Josh Hodas for making my job easier.

The diagrams in this dissertation and in many of my papers were drawn using the macros of Paul Taylor. I thank him for making it so much easier to typeset them.

My work at Penn was support by National Science Foundation grants IRI-86-10617 and IRI-90-04137 and by Army Research Office grant DAAL03-89-C-0031-PRIME. I am grateful to these organizations for providing the funds. Last but not least, I thank the Institute of Systems Science at the National University of Singapore for allowing me a long leave of absence to do my doctoral research.

ABSTRACT

QUERYING NESTED COLLECTIONS

Limsoon Wong

Advisor: Peter Buneman

This dissertation investigates a new approach to query languages inspired by structural recursion and by the categorical notion of a monad.

A language based on these principles has been designed and studied. It is found to have the strength of several widely known relational languages but without their weaknesses. This language and its various extensions are shown to exhibit a conservative extension property, which indicates that the depth of nesting of collections in intermediate data has no effect on their expressive power. These languages also exhibit the finite-cofiniteness property on many classes of queries. These two properties provide easy answers to several hitherto unresolved conjectures on query languages that are more realistic than the flat relational algebra.

A useful rewrite system has been derived from the equational theory of monads. It forms the core of a source-to-source optimizer capable of performing filter promotion, code motion, and loop fusion. Scanning routines and printing routines are considered as part of optimization process. An operational semantics that is a blending of eager evaluation and lazy evaluation is suggested in conjunction with these input-output routines. This strategy leads to a reduction in space consumption and a faster response time while preserving good total time performance. Additional optimization rules have been systematically introduced to cache and index small relations, to map monad operations to several classical join operators, to cache large intermediate relations, and to push monad operations to external servers.

A query system Kleisli and a high-level query language CPL for it have been built on top of the functional language ML. Many of my theoretical and practical contributions have been physically realized in Kleisli and CPL. In addition, I have explored the idea of open system in my implementation. Dynamic extension of the system with new primitives, cost functions, optimization rules, scanners, and writers are fully supported. As a consequence, my

system can be easily connected to external data sources. In particular, it has been successfully applied to integrate several genetic data sources which include relational databases, structured files, as well as data generated by special application programs.

Contents

List of Figures

Part I

The adventure of a logician-engineer

Chapter 1

Introduction

> *We don't really know what the basic equations of physics are, but they have to have great mathematical beauty.* Paul Dirac

The flat relational data model Codd [41] introduced two decades ago is a simple and powerful theory. However, I feel that this time-worn theory is not quite the "right" theory to support modern database applications. The objective of this report is to construct an improved system for querying large collections.

Organization

Section 1.1. A brief description of structural recursion [26] as a paradigm for querying collection types is given. I illustrate its expressive power and efficiency with examples. I then propose a new approach to querying databases based on a very natural restriction of this paradigm.

Section 1.2. This section organizes the four theoretical themes of this report: the study of a query language as a restricted form of structural recursion; the study of its expressive power through the conservative extension property; the study of its expressive power through the

finite-cofiniteness property; and the concretization of an abstract theoretical language into a more realistic query language.

Section 1.3. My theoretical work results in the design of a high-level query language called CPL. Two pure examples from CPL are given to provide a taste of the kind of query languages that my approach leads to and to provide a rarefied picture of its connection with structural recursion.

Section 1.4. This section organizes the four practical themes of this report: the study of optimizations that can be expressed within my query languages; the study of how other kinds of optimizations can be brought in; the empirical verification of the effectiveness of these optimizations; and the construction of a real extensible query system and its application to query heterogenous biomedical data sources.

Section 1.5. My practical work culminates in the implementation of an open query system in ML called Kleisli. CPL is implemented on top of Kleisli and serves as its high-level query language. An example distilled from real genetic queries handled by the system is given to provide a solid demonstration of the achievement of the system in the biomedical database area and to provide an idea of its potential in the broader information integration arena.

1.1 Thesis

Structural recursion

Past experience lead Backus [13] to propose an applicative programming style with a well-chosen collection of primitives. Research on the Bird-Meertens formalism on lists suggests that such a style is remarkably expressive (see Bird [21, 22]; Meertens [143]; and Bird and Wadler [23]). Programming with sets works out the same way (see Codd [41]; Ohori, Buneman, and Tannen [153]; and Bancilhon, Briggs, Khoshafian, and Valduriez [15]). A significant idea in the above work is that they identified certain simple forms of recursion and advocated programming in these restricted forms. One such simple form of recursion is

structural recursion, which I now present based on the work of Tannen and Subrahmanyam [28] and Tannen, Buneman, and Naqvi [26].

As illustrated by Wadler [196], a more abstract view of data types leads to much simpler programs. I thus adopt the abstract view of collection types described below. Informally, an object of type $\lceil s \rfloor$ is a collection of objects of type s, and three operations are expected to be available on objects of type $\lceil s \rfloor$, as depicted in Figure 1.1, where $\lceil \rfloor$ forms the empty

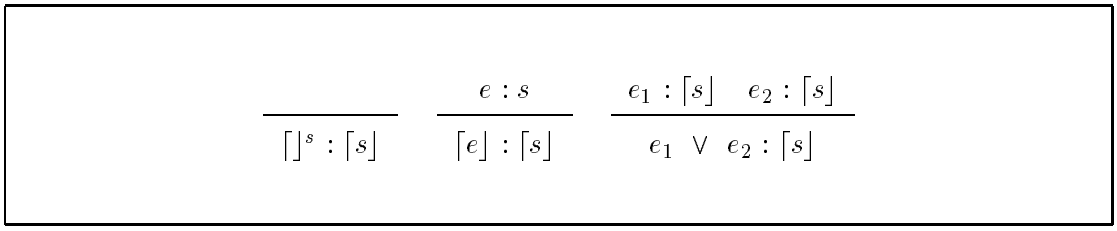


Figure 1.1: The constructors for collection types.

collection, $\lceil e \rfloor$ forms a singleton collection, and $e_1 \vee e_2$ forms a new collection by combining two existing collections. There are many ways to interpret collection types. For example:

- Sets. Interpret $\lceil s \rfloor$ as finite sets of type s; $\lceil \rfloor$ as the empty set; $\lceil e \rfloor$ as the singleton set containing e; and $e_1 \vee e_2$ as union of sets e_1 and e_2. In this case $\vee$ is associative, commutative, and idempotent; and $\lceil \rfloor$ is the identity for $\vee$.

- Bags. Interpret $\lceil s \rfloor$ as finite multisets (also known as bags) of type s; $\lceil \rfloor$ as the empty bag; $\lceil e \rfloor$ as the singleton bag containing e; and $e_1 \vee e_2$ as additive union of bags e_1 and e_2. In this case $\vee$ is associative, commutative, but not idempotent; and $\lceil \rfloor$ is the identity for $\vee$.

- Lists. Interpret $\lceil s \rfloor$ as finite lists of type s; $\lceil \rfloor$ as the empty list; $\lceil e \rfloor$ as the singleton list containing e; and $e_1 \vee e_2$ as concatenation of lists e_1 and e_2. In this case $\vee$ is associative but is neither commutative nor idempotent; and $\lceil \rfloor$ is the identity for $\vee$.

- Other possibilities include orsets, certain kinds of tree, finite maps, arrays, etc. See Libkin and myself [131]; Watt and Trinder [199]; Atkinson, Richard, and Trinder [10]; and Buneman [32].

Tannen and Subrahmanyam [28] described one way, depicted in Figure 1.2, of doing structural recursion of the above view of collection types that is inspired by the notion of universal property.

$$\frac{u : t \times t \to t \quad f : s \to t \quad i : t}{sru(u, f, i) : \lceil s \rfloor \to t}$$

obeying the following three axioms:

$$sru(u, f, i)\lceil\rfloor = i$$

$$sru(u, f, i)\lceil e \rfloor = f(e)$$

$$sru(u, f, i)(e_1 \ \vee \ e_2) = u(sru(u, f, i)(e_1), \ sru(u, f, i)(e_2))$$

Figure 1.2: The structural recursion construct for collection types.

The need for $sru(u, f, i) : \lceil s \rfloor \to t$ to respect the three axioms means that it is not well defined on every u, f, and i, depending on the interpretation of $\lceil s \rfloor$. Let me use an example of Val Tannen to illustrate this point. Consider the function defined as $Card \triangleq sru(+, \ id, \ 0)$, where $+$ is addition, and id is the identity function. If $\lceil \mathbb{N} \rfloor$ is interpreted as lists (or bags) of natural numbers, then $Card : \lceil \mathbb{N} \rfloor \to \mathbb{N}$ is the cardinality function on list (or bag). However, here is what happens when $\lceil \mathbb{N} \rfloor$ is interpreted set-theoretically so that $\vee$ is idempotent: $1 = Card\lceil 4 \rfloor \ = Card(\lceil 4 \rfloor \ \vee \ \lceil 4 \rfloor) = 1 + 1 = 2$. That is, the equational theory has become inconsistent. The reason for this is that $+$ is not idempotent while $\vee$ under our set-theoretic interpretation is.

Therefore, some restrictions must be placed on $sru(u, f, i)$ to ensure that it is well defined for a given interpretation of $\lceil s \rfloor$. A set of simple conditions guaranteeing the well-definedness

of structural recursion with respect to lists, bags, and sets was worked out by Tannen and Subrahmanyam [28].

Proposition 1.1.1 *$sru(u, f, i) : \lceil s \rfloor \to t$ is well defined when $\lceil s \rfloor$ is interpreted list-theoretically, bag-theoretically, or set-theoretically, if (t, u, i) is respectively a monoid, a commutative monoid, or a commutative idempotent monoid.* □

Examples

Let me illustrate the expressive power and efficiency of structural recursion by some examples on *sets*, taken mostly from Tannen, Buneman, and Naqvi [26]. I begin with some common operators for sets. These examples can also be defined in first-order logic. They are chosen to illustrate the mileage that can be obtained using structural recursion of the simple form $sru(\vee, f, \lceil \rfloor)$.

- The function $map(f) : \lceil s \rfloor \to \lceil s \rfloor$, where $f : s \to t$, such that $map(f)\{o_1, \ldots, o_n\}$ is the set $\{f(o_1), \ldots, f(o_n)\}$ is definable as $map(f) \triangleq sru(\vee,\ F,\ \lceil \rfloor)$, where F is the function such that $F(x) = \lceil f(x) \rfloor$.

- The function $select(p) : \lceil s \rfloor \to \lceil s \rfloor$, where $p : s \to \mathbb{B}$ is a predicate, such that $select(p)(O)$ is the largest subset of O whose elements satisfy p. It is definable as $select(p) \triangleq sru(\vee, F, \lceil \rfloor)$, where F is the function such that $F(x) = \mathit{if}\ p(x)\ \mathit{then}\ \lceil x \rfloor\ \mathit{else}\ \lceil \rfloor$.

- The function $flatten : \lceil \lceil s \rfloor \rfloor \to \lceil s \rfloor$ which flattens a set of sets is definable as $flatten \triangleq sru(\vee,\ id, \lceil \rfloor)$, where id is the identity function.

- The function $pairwith_2 : s \times \lceil t \rfloor \to \lceil s \times t \rfloor$ so that $pairwith_2(o, \{o_1, \ldots, o_n\}) = \{(o, o_1), \ldots,\ (o, o_n)\}$ is definable as $pairwith_2(o, O) \triangleq sru(\vee, F, \lceil \rfloor)(O)$, where F is the function such that $F(x) = \lceil (o, x) \rfloor$. The analogous function $pairwith_1 : \lceil s \rfloor \times t \to \lceil s \times t \rfloor$ can also be so defined.

- The function $cartprod : \lceil s \rfloor \times \lceil t \rfloor \to \lceil s \times t \rfloor$ such that $cartprod(O_1, O_2)$ is the cartesian product of O_1 and O_2 is definable as $cartprod \triangleq flatten \circ map(pairwith_1) \circ pairwith_2$.

- The function $\# : \lceil r \times s \rfloor \times \lceil s \times t \rfloor \to \lceil r \times t \rfloor$ such that $O_1 \# O_2$ is the relational composition of O_1 and O_2 is expressible as $\# \triangleq sru(\vee,\ F,\ \lceil \rfloor) \circ cartprod$, where $F((x,y),(u,v)) = \mathit{if}\ y = u\ \mathit{then}\ \lceil (x,v) \rfloor\ \mathit{else}\ \lceil \rfloor$.

- The function $member : s \times \lceil s \rfloor \to \mathbb{B}$ such that $member(o, O)$ is true if and only if o is a member of O is definable as $member(o, O) \triangleq sru(or,\ F,\ \mathit{false})(O)$, where $F(x) = (x = o)$.

Tannen and Subrahmanyam [28] had a second form of structural recursion $sri(h, i) : \lceil s \rfloor \to t$, where $h : s \times t \to t$ and $i : t$. It can be defined, with the help of some higher-order functions, in terms of the first form of structural recursion as $sri(h, i)(l) = sru(U,\ I,\ id)(l)(i)$, where $U(x, y)(z) = x(y(z))$ and $I(x)(y) = h(x, y)$. (A more complicated but purely first-order implementation of *sri* in terms of *sru* has also been discovered. See Suciu and Wong [180].) I use this second form of structural recursion to give a few more examples. These examples are queries which are known to be inexpressible in first-order logic [7, 38, 39]. They illustrate the power and flexibility of structural recursion.

- The function $tc : \lceil s \times s \rfloor \to \lceil s \times s \rfloor$ such that $tc(O)$ is the transitive closure of O is definable as $tc \triangleq sri(F,\ \lceil \rfloor)$, where $F(o, O) = \lceil o \rfloor \vee\ O\ \vee\ (\lceil o \rfloor \# O) \vee\ (O \# \lceil o \rfloor) \vee (O \# \lceil o \rfloor\ \# O)$.

- The function $odd : \lceil s \rfloor \to \mathbb{B}$ such that $odd(O)$ is true if and only if the cardinality of O is odd is definable: $odd \triangleq \pi_2 \circ sri(F,\ (\lceil \rfloor,\ \mathit{false}))$, where $F(x, (y, z)) = \mathit{if}\ member(x, y)\ \mathit{then}\ (y, z)\ \mathit{else}\ (\lceil x \rfloor \vee\ y,\ \mathit{not}\ z)$.

- The function $Card : \lceil s \rfloor \to \mathbb{N}$ such that $Card(O)$ is the cardinality of the set O is expressible: $Card \triangleq \pi_2 \circ sri(F,\ (\lceil \rfloor,\ 0))$, where $F(x, (y, z)) = \mathit{if}\ member(x, y)\ \mathit{then}\ (y,\ z)\ \mathit{else}\ (\lceil x \rfloor \vee\ y,\ 1 + z)$.

All the examples above execute in polynomial time with respect to the size of input with an appropriate implementation of the functions $sru(u, f, i)$. For a discussion on the efficiency of the transitive closure example, see Tannen, Buneman, and Naqvi [26]. Now, let me provide an expensive example by using *sru* to compute powerset.

- The function $powerset : \lceil s \rfloor \to \lceil\lceil s \rfloor\rfloor$ which computes the powerset of its input is definable: $powerset \triangleq sru(map(\cup) \circ cartprod,\ F,\ \lceil\lceil\rfloor\rfloor)$, where $F(x) = \lceil\lceil\rfloor\rfloor \vee \lceil\lceil x \rfloor\rfloor$.

My final example is designed to demonstrate the possibility of applying structural recursion to query nested relations. This example also uses only recursion of the form $sru(\vee, \cdot, \lceil\rfloor)$.

- The function $nest_2 : \lceil s \times t \rfloor \to \lceil s \times \lceil t \rfloor\rfloor$ which expresses the relational nesting on its input is definable: $nest_2(O) \triangleq sru(\vee,\ F,\ \lceil\rfloor)(O)$, where $G(x)(u, v) = \mathit{if}\ x = u\ \mathit{then}\ \lceil v \rfloor\ \mathit{else}\ \lceil\rfloor$ and $F(x, y) = \lceil(x,\ sru(\vee,\ G(x),\ \lceil\rfloor)(O))\rfloor$.

Towards monads

As seen earlier, structural recursion is a rather attractive paradigm for querying lists, bags, and sets. It has considerable expressive power; it is relatively efficient; it scales from flat collections to nested collections. The only caveat is the need to verify certain preconditions for well-definedness. Automatic verification of these conditions, as discussed in Tannen and Subrahmanyam [28], is very hard and the general problem is undecidable.

One way to proceed from here is to equip the compiler with a powerful theorem prover. The compiler accepts and compiles only those programs whose definedness can be verified by the theorem prover. This approach was proposed by Immerman, Patnaik, and Stemple [104]. This approach is feasible [175], but it requires extensive experience with theorem provers [173].

Another way is to abandon structural recursion and look for alternative primitives which have similar expressive power and performance. Powerset operators, fixpoint operators, and

while-loops are possible alternatives. They were already competently and fruitfully studied in Abiteboul and Beeri [2], Hull and Su [99], Grumbach and Milo [77], Grumbach and Vianu [79], Gyssens and Van Gucht [83], Kuper [123], etc.

In the two approaches above, there is sufficient power to express non-polynomial time computation. Their aim is to retain as much expressive power of structural recursion as possible. In the context of querying databases, it is reasonable to limit queries to those which are practical. Therefore, a third approach can be envisioned. I propose to impose further syntactic restrictions so that any expressions conforming to these restrictions are automatically well defined. Moreover, I propose that these restrictions should be sufficiently strong to limit queries to those which have polynomial time and data complexity. This third approach was taken by Tannen, Buneman, and Naqvi [26]. They found some syntactic restrictions which cut structural recursion down to a language whose expressive power is that of the traditional relational algebra [41]. In this report, a simpler restriction is considered: queries are expressed using only structural recursion of the form $sru(\vee, \cdot, \lceil\rfloor)$.

By restricting structural recursion to $sru(\vee, \cdot, \lceil\rfloor)$, the u and i parameters of $sru(u, f, i)$ are fixed and while f is allowed to vary. The restriction $sru(\vee, f, \lceil\rfloor)$ is very natural. It respectively cuts structural recursion on lists, bags, and sets down to homomorphisms of monoids (of lists with concatenation as the binary operation and the empty list as identity), commutative monoids (of bags with additive bag union as the binary operation and the empty bag as identity), and commutative idempotent monoids (of sets with union as the binary operation and the empty set as identity). Judging from the examples given earlier, it is very promising in terms of expressive power and efficiency. Tannen, Buneman, and I [29] showed that this form of structural recursion corresponds to the categorical notion of a monad, thus providing a basis for constructing algebras and calculi suitable for abstract manipulation of collections. Encouraged by these observations, I propose to construct query languages for collection types around this syntactic restriction on structural recursion.

1.2 Overview of theoretical results

There are four theoretical themes in this dissertation. The results of my investigation on these themes are organized into the following four chapters, one for each theme.

Querying nested relations

The theme of Chapter 2 is the design of query languages based on the restriction of structural recursion proposed in Section 1.1. I have considered this theme for conventional collection types such as sets in a paper with Tannen and Buneman [29] and bags in a paper with Libkin [130], as well as unconventional collection types such as orsets [101, 102] in another paper with Libkin [131]. I discuss here only the query language for sets I have thus obtained. The set-theoretic interpretation of my restricted structural recursion language is denoted by $\mathcal{NRC}$. The main results are:

- A language $\mathcal{NRC}$ based on restricting structural recursion on sets to $sru(\cup, f, \{\})$ is presented. A fully algebraic version of $\mathcal{NRC}$, based on a more abstract presentation of monads, is given as well. Functions definable in the algebra are shown to have polynomial time complexity. The equivalence between these two formulations are sketched. An interesting aspect of the proof is that it is largely equational, in contrast to the usual semantic proofs of this kind of results.

- Variants are a useful data modeling concept [100] and are ubiquitous in modern programming languages [82]. I describe how they can be added to $\mathcal{NRC}$. I show that variants do not change the expressive power of $\mathcal{NRC}$ in any essential way. A corollary of this result is that adding booleans and the conditional construct to $\mathcal{NRC}$ does not greatly affect its expressive power. The most interesting aspect of this result is its entirely equational proof.

- All common non-monotonic operators such as the equality test, the membership test, the subset test, set intersection, set difference, and relational nesting are inter-

definable using $\mathcal{NRC}$ as the ambient language. Since adding such operators to $\mathcal{NRC}$ does not take it out of polynomial time, this result strengthens a similar result of Gyssens and Van Gucht [84] who proved the inter-definability of these operators in the presence of the costly *powerset* operator. For this reason, I use $\mathcal{NRC}(\mathbb{B},=)$ as my ambient language.

- $\mathcal{NRC}(\mathbb{B},=)$ is shown to possess precisely the same expressive power as the well-known nested relational algebra of Thomas and Fischer [183]. Then I argue that $\mathcal{NRC}(\mathbb{B},=)$ can be profitably regarded as the "right" core for nested relational languages.

Conservative extension properties

The theme of Chapter 3 is the conservative extension property of query languages. If a query language possesses the conservative extension property, then the class of functions having certain input and output heights (that is, the maximal depth of nesting of sets in the input and output) definable in the language is independent of the height of intermediate data used. Such a property can be used to prove interesting expressibility results. The main results are:

- $\mathcal{NRC}(\mathbb{B},=)$ has the conservative extension property. Paredaens and Van Gucht [159] proved a similar result for the special case when input and output are flat relations. Their result was complemented by Hull and Su [99] who demonstrated the failure of independence when the *powerset* operator is present and input and output are flat. The theorem of Hull and Su was generalized to all input and output by Grumbach and Vianu [79]. My result generalizes Paredaens and Van Gucht's to all input and output, providing a counterpart to the theorem of Grumbach and Vianu. A corollary of this result is that $\mathcal{NRC}(\mathbb{B},=)$, when restricted to flat relations, has the same power as the flat relational algebra [41].

- As a result $\mathcal{NRC}(\mathbb{B},=)$ cannot implement some aggregate functions found in real database query languages such as the "select average from column" of SQL [106]. I

therefore endow the basic nested relational language with rational numbers, some basic arithmetic operations, and a summation construct. The augmented language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is then shown to possess the conservative extension property. This result is new because conservativity in the presence of aggregate functions had never been studied before.

- $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is augmented with a linear order on base types. It is then shown that the linear order can be lifted within $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ to every complex object type. The augmented language also has the conservative extension property. This fact is then used to prove a number of surprising results. As mentioned earlier, Grumbach and Vianu [79] and Hull and Su [99] proved that the presence of *powerset* destroys conservativity in the basic nested relational language. A corollary of my theorem shows that this failure can be repaired with a little arithmetic operations, aggregate functions, and linear orders.

- A notion of internal generic family of functions is defined. It is shown that the conservative extension property of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ endowed with (well-founded) linear orders can be preserved in the presence of any such family of functions. This result is a deeper explanation of the surprising conservativity of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ in the presence of *powerset* and other polymorphic functions.

Finite-cofinite properties

Predicates definable in first-order logic exhibits a finite-cofinite property. That is, they either hold for finitely many things or they fail for finitely many things. The theme of Chapter 4 is the finite-cofiniteness property in the various extensions of my basic query language. The main results are:

- Every property expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ on rational numbers is shown either to hold for finitely many rational numbers or to fail for finitely many rational numbers. This result generalizes the above mentioned property of first-order

logic. A corollary of this result is that, inspite of its arithmetic power, $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ cannot test whether one number is bigger than another number. This justifies the augmentation of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ with linear orders on base types.

- Every property expressible in the augmented language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ on natural numbers is shown to be finite or cofinite. Many consequences follow from this result, including the inexpressibility of parity test in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ on natural numbers. This is a very strong evidence that the conservative extension theorem for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ is not a consequence of Immerman's result on fixpoint queries in the presence of linear orders.

- Properties on certain classes of graphs in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ when the linear order is restricted to rational numbers is considered. I show that these properties are again finite-cofinite. This result settles the conjectures of Grumbach and Milo [77] and Paredaens [158] that parity-of-cardinality test, transitive closure, and balanced-binary-tree test cannot be expressed with aggregate functions or with bags. This also generalizes the classic result of Aho and Ullman [7] that flat relational algebra cannot express transitive closure to a language which is closer in strength to SQL.

Towards a practical query language

The theme of Chapter 5 is the realization of an abstract language like $\mathcal{NRC}(\mathbb{B}, =)$ into a real query language called CPL. The outstanding features of CPL worth mentioning here are:

- A rich data model is supported. In particular, sets, lists, bags, records, and variants can be freely combined. The language itself is obtained by orthogonally combining constructs for manipulating these data types.

- A comprehension syntax is used to uniformly manipulate sets, lists, and bags. CPL's comprehension notation is a generalization of the list comprehension notation of functional languages like Miranda [188].

- A pattern matching mechanism is supported. In particular, convenient partial-record patterns and variable-as-constant patterns are supported. The former is also available in languages like Machiavelli [153] but not in languages like ML [144]. The latter is not available elsewhere at all.

- Types are automatically inferred. In particular, CPL has polymorphic record types. However, the type inference system is simpler than that of Ohori [154], Remy [165], etc.

- Easily extensible. External functions can be easily added to CPL. New data scanners and new data writers can be easily added to CPL. Thus CPL is readily connected to different external systems.

- An extensible optimizer is available. The basic optimizer does loop fusion, filter promotion, and code motion. It optimizes scanning and printing of external files. It has been extended to deal with joins by picking alternative join operators and by migrating them to external servers. Additional optimization rules can be introduced readily.

1.3 Prelude to real applications

The simple restricted form of structural recursion that I explore in this dissertation leads to a rather appealing query language and system. The query language is called CPL. It is previewed in this section via two example queries which illustrate its flavor. I then illustrate its connection to structural recursion by explaining how CPL handles the second example.

is a query to find employees who are allocated an office O in a building X. It can be applied to any database DB having at least columns `#emp`, `#room`, and `#bldg`. It produces a set of employees.

```
primitive inOfficeOInBuildingX == (\DB, \O, \X) =>
  { E | (#room: O, #bldg: X, #emp: \E, ...) <- DB };
Result : primitive inOfficeOInBuildingX registered.
Type   : (#1:{(#emp:''1,#bldg:''2,#room:''3; ''4)},#2:''3,#3: ''2)
         ->{''1}
```

Let me use this example to explain some relevant part of CPL syntax. (A dialect of it can be found in Buneman, Libkin, Suciu, Tannen, and Wong [31].) An expression of the form p `=>` e defines a function whose input is required to match the pattern p and whose output is computed by the expression e. In the above example, the input pattern p is `(\DB, \O, \X)`, which specifies that the input must be a triple. Prefixing an identifier in a pattern with a slash is CPL's way of introducing a new variable. Hence this pattern introduces three new variables DB, O, and X, which bind respectively to the first, second, and third components of the input when the function is applied. In the example, the expression corresponding to e has the form of a set comprehension.

A set comprehension of the form $\{e_1 \mid q$ `<-` $e_2\}$ means perform e_1 on every element of e_2 that matches the pattern q and then union the results into a set. In the example above, e_2 is the value which DB is bound to (that is, the first component of the input to the function). Here the pattern q is `(#room: O, #bldg: X, #emp: \E, ...)`, which matches records having at least fields `#room`, `#bldg`, and `#emp`. (If the ellipsis is omitted from the pattern, an exact match is then required.) Moreover, this pattern introduces a new variable E which is bound to the value associated with the `#emp` field of the record. Notice that O and X are not slashed in this pattern. That means they are not new variables being introduced. Rather, it means the value associated with the `#room` field must match whatever

O is currently bound to (in this case, to the second component of the input to the function) and the value associated with the `#bldg` field must match whatever X is currently bound to (in this case, to the last component of the input to the function). Finally, the e_1 part of the comprehension is the expression E, which corresponds to the value at the `#emp` field of the pattern q. Hence as the pattern q is being matched against each element of e_2, e_1 extracts the names of employees.

Assuming we have the following database file of office allocations:

```
readfile DB from "OffAlloc" using StdIn;
Result : File DB registered.
Type   : {(#room:string, #bldg:string, #emp:string, #phone:int)}

DB;
Result : {(#phone:85879, #emp:"limsoon", #bldg:"moore",  #room:"062"),
          (#phone:85879, #emp:"jong",    #bldg:"moore",  #room:"062"),
          (#phone:85842, #emp:"jinah",   #bldg:"moore",  #room:"060"),
          (#phone:85842, #emp:"ben",     #bldg:"moore",  #room:"060"),
          (#phone:83224, #emp:"chuck",   #bldg:"pender", #room:"132")}
Type   : {(#room:string, #bldg:string, #emp:string, #phone:int)}
```

Then we can check who has been given room 062 in the Moore Building (the `@`-sign is CPL's symbol for function application):

```
inOfficeOinBuildingX @ (DB, "062", "moore");
Result : {"limsoon", "jong"}
Type   : {string}
```

is a query to group together employees who share an office in a given building X. It is expressed in CPL in the rather simple manner below. It can be applied to any database DB having at least columns `#emp`, `#room`, and `#bldg`. It produces a nested relation having columns `#room` and `#occupants`, where entries in the latter column are sets themselves.

```
primitive shareOfficeInBuildingX == (\DB, \X) =>
   { (#room   : 0,
      #occupants: { E | (#room: 0, #bldg: X, #emp: \E, ...) <- DB })
    | (#room: \0, #bldg: X, ...) <- DB };
Result : Primitive shareOfficeInBuildingX registered.
Type   : (#1:{(#emp:''39, #bldg:''10, #room:''20; ''5)}, #2:''10)
         ->{(#occupants:{''39}, #room:''20)}
```

Then we can check who shares an office with whom in the Moore Building:

```
shareOfficeInBuildingX @ (DB, "moore");
Result : {(#occupants: {"limsoon", "jong"}, #room: "062"),
          (#occupants: {"jinah", "ben"}, #room: "060")}
Type   : {(#occupants:{string}, #room:string)}
```

Let me try to reveal the connection of CPL to structural recursion $sru(\cup, f, \{\})$ by explaining how the query `shareOfficeInBuildingX` is handled in CPL in the absence of optimization. The first step taken by the CPL compiler is to replace certain enhanced forms of pattern matching by simpler patterns. Thus the query becomes:

```
(\DB, \X) =>
   {(#room: 0,
     #occupants: {E | (#room: \0', #bldg: \X', #emp: \E, ...) <- DB,
                      0' = 0, X' = X })
```

```
    | (#room: \0, #bldg: \X'', ...) <- DB, X'' = X };
```

The simple patterns are then removed so that the query becomes a pure set comprehension:

```
\Y  =>
   {(#room: A.#room,
     #occupants:{B.#emp | \B <- Y.#1, B.#room=A.#room, B.#bldg=Y.#2})
   | \A <- Y.#1, A.#bldg = Y.#2 };
```

Finally, set comprehensions are implemented in terms of the primitive `sext`$\{e_1 \mid \backslash x$ `<-` $e_2\}$, which is precisely our restricted structural recursion $sru(\cup, \lambda x.e_1, \{\})(e_2)$.

```
\Y => sext{ if A.#bldg = Y.#2
            then {(#room: A.#room,
                   #occupants:  sext{ if B.#room = A.#room
                                      then if B.#bldg = Y.#2
                                           then {B.#emp}
                                           else {}
                                      else {} | \B <- Y.#1}
            else { }    | \A <- Y.#1};
```

1.4 Overview of practical results

There are four practical themes in this dissertation. My work on these these themes is organized into the following four chapters, one for each theme.

'Monadic' optimizations

Query evaluation has three phases: input, evaluation, output. Query cost has three aspects: total time, response time, and peak memory usage. The theme of Chapter 6 is the investiga-

tion of techniques for improving queries over nested collections which takes all three phases of evaluation and all three aspects of cost into account. In particular, I consider techniques that are expressible in query languages based on my restricted form of structural recursion. The main contributions are:

- Structural rewrite rules of sufficient generality to capture fusion of loops, migration of filters, etc. in the pure language $\mathcal{NRC}$ are given. Most of these rules come directly from orientating the axioms of $\mathcal{NRC}$ in a suitable way which eliminates large intermediate data. In fact, they form a superset of the rules used in proving the conservative extension property for $\mathcal{NRC}(\mathbb{B}, =)$.
- The input phase is abstracted as a process of converting input stream into a complex object. Scanning constructs are introduced. Rewrite rules for exploiting these constructs to reduce excessive space consumption caused by loading entire files are given.
- The output phase is abstracted as a process for converting a complex object into an output stream. Printing constructs are introduced. A lazy operational semantics is suggested for these constructs. Rewrite rules for exploiting these constructs to reduce space consumption and to improve response time are given. It is interesting to note that query execution is eager by default and laziness is introduced by these rewrite rules. This strategy is in contrast to the tradition of lazy languages where execution is lazy by default and eagerness is introduced by performing strictness analysis [97].

Additional optimizations

There exists a large body of literature on optimization in flat relational system. The theme of Chapter 7 is to investigate how some of these optimizations can be applied to my languages. Flat relational optimizations that I have generalized to my languages are enumerated below.

- Two new constructs are introduced to cache and to index small external relations into memory. Rules are suggested for using these new operators in query optimization. An-

other new construct is introduced to cache large intermediate data onto disk to avoid recomputation. Rules are given for using this new construct in query optimization.

- A new construct is introduced to capture the blocked nested-loop join algorithm. Rules for recognizing whether a nested loop is a join or not and for other general optimizations involving this new construct are given. Another new construct is introduced to capture the indexed blocked-nested-loop join algorithm. Rules for recognizing whether a join condition in a blocked nested-loop join can be indexed or not and for other general optimizations involving this new construct are given.

- A new construct is introduced to illustrate the use of relational servers as providers of external data. Rules for moving selection, projection, and join operations to these servers are given. Another new construct is introduced to illustrate the use of nonrelational servers as providers of external data. Rules for moving selection and flattening operations to these servers are given.

Performance and experiments

I have implemented many of the optimizations outlined earlier. Several experiments were performed in part to check the correctness of my implementation and in part to validate the effectiveness of these optimizations. Chapter 8 is a summary of some of these experiments. The results support the expectation that the optimizations I have implemented are indeed optimizations.

Towards a useful query system

I have built an open query system Kleisli and have implemented the collection programming language CPL, as a high-level query language for it. (Kleisli is just a library of routines in a host programming language. It is itself neither a query language nor a programming language. CPL is a particular high-level syntax for manipulating collections. This syntax is interpreted in terms of the routines provided in Kleisli. In other words, CPL is a query

language for Kleisli.) The openness of Kleisli allows the easy introduction of new primitives, optimization rules, cost functions, data scanners, and data writers. Furthermore, queries that need to freely combine external data from different sources are readily expressed in CPL. I claim that Kleisli, together with CPL, is a suitable tool for querying heterogenous data sources. Chapter 9 presents an overview of Kleisli and several examples towards this claim.

- An extended example is used illustrate the libraries provided in Kleisli for application programming and for building new primitives. The example is the implementation of the indexed blocked-nested-loop join operator [145].
- Three examples are used to illustrate the ease of adding new data scanners to Kleisli. Specifically, I show how a driver for Sybase servers, a driver for ASN.1 [151] servers, and a sequence similarity package are introduced into Kleisli and CPL.
- Two examples are presented to illustrate the ease of writing new optimization rules for the extensible optimizer of Kleisli. I show how to describe a rule for turning a blocked nested-loop join into an indexed blocked-nested-loop join and a rule for pushing join operations on external data to their source servers.

1.5 A real application to query genetic databases

Kleisli [89] is a query system whose most outstanding feature is its openness: new primitives, new optimization rules, new cost estimation functions, new data readers and writers can all be dynamically added to the system. The collection programming language CPL has been built on top of Kleisli and serves as its high-level query language. The openness of Kleisli allows easy connection to several genetic databases and their associated tools. A partial list of these databases and tools include GDB [161], NCBI ASN.1 [151], Sortez [88], Entrez [150], and BLAST [9]. These can then be freely combined in any CPL queries.

In Spring 1993, the Department of Energy published a report [57] which listed twelve

"impossible" genomic data retrieval problems. These were thought to be impossible because they involve the integration of databases, structured files, and applications — something well beyond the capabilities of any existing heterogenous database system. A colleague from the genetic department at Penn and I have succeeded in implementing many of these hard queries using CPL. I now present an extended example to demonstrate the possibility of using CPL as a general query language for genetic databases.

The example

is the following problem, which is quite typical of the so-called impossible queries:

> Find information on the known DNA sequences on chromosome 22, as well as information on homologous sequences in this area.

To tackle this problem, access to GDB, Sortez, and Entrez is needed. GDB is the main Sybase relational database. I use it for obtaining marker information for the region in question. This database is located in Baltimore and has to be accessed remotely. Entrez is a special collection of tools for the NCBI ASN.1 database. I use it for accessing precomputed links to retrieve homologous sequences. This database is stored on a CD-ROM connected directly to my machine at Penn's computing department. As GDB and Entrez use different identifiers, a third database is needed to look up the alternative names. I use Sortez, a home-brew Sybase derivative of the MEDLINE portion of NCBI ASN.1, for this purpose. Sortez is located at Penn's genetic department and has to be accessed remotely. All three of them are available in CPL as external primitives.

The primitive for accessing GDB

is the function `GDB`. This function takes in a string. It sends this string as a Sybase query to the GDB server. The result is then returned as a set of records of the appropriate type. See Section 9.3 for its implementation.

Our example requires us to retrieve from GDB genetic records within a certain range. This is accomplished by defining a new primitive `Loci22` in terms of `GDB` as below.

```
primitive Loci22 == GDB @
  "select distinct
      genbank_ref, locus_symbol, loc_cyto_chrom_num,
      rtrim(loc_cyto_band_start)+'-'+rtrim(loc_cyto_band_end),
      loc_cyto_band_start_sort, loc_cyto_band_end_sort
   from
      locus, locus_cyto_location, object_genbank_eref
   where
      locus.locus_id=object_genbank_eref.object_id and
      object_genbank_eref.object_id = locus_cyto_location.locus_id
      and object_class_key = 1 and loc_cyto_chrom_num = '22'
      order by loc_cyto_band_start_sort,loc_cyto_band_end_sort";
```

When `Loci22` is invoked, a set of records beginning with the following two is returned:

```
{(#genbank_ref: "M15492", #locus_symbol: "D22Z2",
  #loc_cyto_chrom_num: "22", #bogus4: "cen-",
  #loc_cyto_band_start_sort: 220008, #loc_cyto_band_end_sort: 220008),
 (#genbank_ref: "M15493", #locus_symbol: "D22Z2",
  #loc_cyto_chrom_num: "22", #bogus4: "cen-",
  #loc_cyto_band_start_sort: 220008, #loc_cyto_band_end_sort: 220008),
  ...}
```

The primitive for accessing Sortez

is the function `Sortez`. This function takes in a string. It sends this string to the Sortez server as a Sybase query. The result is returned as a set of records. See Section 9.3 for its implementation.

Our example requires us to define a function `CurrentACC` which takes in a GenBank reference and returns all its alternative identifiers. This function is implemented by looking the aliases up in Sortez as follows: (1) it takes in a string `x`, (2) it appends `x` to the string `select locus, accession, title, length, taxname from gb_head_accs where pastaccession =` to form a Sybase query, and (3) passes the query to `Sortez`. (The symbol `o` is CPL's symbol for function composition. The `^` sign is the string concatenation operator.)

```
primitive CurrentACC == Sortez o
   (\x =>  "select locus, accession, title, length, taxname
            from gb_head_accs
            where pastaccession = '" ^ x ^ "'") ;
```

For example, `CurrentACC @ "M15492"` returns the singleton set below.

```
{(#locus: "HUMAREPBG", #accession: "M15492",
  #length: 171, #taxname: "",
  #title: "Human alphoid repetitive DNA repeats 1'' monomer,
           clone alpha-RI(680) 22-73-I-1.")}
```

The primitive for accessing Entrez

is the function `EntrezLinks`, which takes in an identifier string and returns a set of genes that are within a certain homological distance of the gene identified by the input string. See Section 9.3 for its implementation.

For example, `EntrezLinks @ "M15492"` gives us the following set of records:

```
{(#ncbi_id: 64, #linkacc: "M22286", #locus: "HUMAREPCI",
  #title: "Human alphoid repetitive DNA repeats 1' monomer,
           clone alpha-X(1020) 22-133 III."),
```

```
(#ncbi_id: 63, #linkacc: "M22278", #locus: "HUMAREPCA",
 #title: "Human alphoid repetitive DNA repeats 1' monomer,
          clone alpha-T(1360) 14-204 III."),
(#ncbi_id: 63, #linkacc: "M22270", #locus: "HUMAREPBS",
 #title: "Human alphoid repetitive DNA repeats 1' monomer,
          clone alpha-T(1360) 14-12 III."),
(#ncbi_id: 61, #linkacc: "M22294", #locus: "HUMAREPCQ",
 #title: "Human alphoid repetitive DNA repeats 1' monomer,
          clone alpha-T(1360) 22-7 III."),
(#ncbi_id: 47, #linkacc: "M81230", #locus: "HUMASATAB",
 #title: "Human alpha satellite DNA sequence.")}
```

The CPL query implementing the example

For the purpose of clarity, let me first define a primitive for extracting similar genes.

```
primitive Homologs == \id =>
  {(x, EntrezLinks @ y) |
         \x & (#accession: \y,...) <- CurrentACC @ id };
```

The meaning of this query is as follow. Given input identifier `id`, iterate over the set `CurrentACC @ id`. Bind `x` to each successive record. Bind `y` to the `#accession` field of the record. Return the pair `(x, EntrezLinks @ y)` at each iteration. Therefore, this query returns all data that are similar to the gene identified by `id` and groups the data with respect to its aliases.

As `EntrezLinks @ y` returns a set, the output of this query is a nested relation. Indeed, `Homologs @ "M15492"` gives the expected singleton set below, where the second field of the single record in the output is itself a set of records.

```
{(#1: (#locus: "HUMAREPBG", #accession: "M15492",
```

```
      #length: 171, #taxname: "",
      #title: "Human alphoid repetitive DNA repeats 1'' monomer,
               clone alpha-RI (680) 22-73-I-1."),
  #2: {(#ncbi_id: 64, #linkacc: "M22286", #locus: "HUMAREPCI",
       #title: "Human alphoid repetitive DNA repeats 1' monomer,
                clone alpha-X(1020) 22-133 III."),
       (#ncbi_id: 63, #linkacc: "M22278", #locus: "HUMAREPCA",
       #title: "Human alphoid repetitive DNA repeats 1' monomer,
                clone alpha-T(1360) 14-204 III."),
       (#ncbi_id: 63, #linkacc: "M22270", #locus: "HUMAREPBS",
       #title: "Human alphoid repetitive DNA repeats 1' monomer,
                clone alpha-T(1360) 14-12 III."),
       (#ncbi_id: 61, #linkacc: "M22294", #locus: "HUMAREPCQ",
       #title: "Human alphoid repetitive DNA repeats 1' monomer,
                clone alpha-T(1360) 22-7 III."),
       (#ncbi_id: 47, #linkacc: "M81230", #locus: "HUMASATAB",
       #title: "Human alpha satellite DNA sequence.")})}
```

The function `Homologs` can now be used as a subquery in the final solution to our problem. We just apply it to each record returned by `Loci22` using a simple comprehension as below:

```
{ (x, Homologs @ id) | \x & (#genbank_ref: \id, ...) <- Loci22 };
```

The result is a nested relation of nested relations. For completeness, the first record of the output of this query is display below.

```
{(#1:(#genbank_ref:"M15492", #locus_symbol:"D22Z2",
      #loc_cyto_chrom_num:"22", #bogus4:"cen-",
      #loc_cyto_band_start_sort:220008,
      #loc_cyto_band_end_sort:220008),
  #2:{(#1:(#locus:"HUMAREPBG", #accession:"M15492",
```

```
            #length:171, #taxname:"",
            #title:"Human alphoid repetitive DNA repeats 1'' monomer,
                    clone alpha-RI(680) 22-73-I-1."),
        #2:{(#ncbi_id:64, #linkacc:"M22286", #locus:"HUMAREPCI",
             #title:"Human alphoid repetitive DNA repeats 1' monomer,
                    clone alpha-X(1020) 22-133 III."),
            (#ncbi_id:63, #linkacc:"M22278", #locus:"HUMAREPCA",
             #title:"Human alphoid repetitive DNA repeats 1' monomer,
                    clone alpha-T(1360) 14-204 III."),
            (#ncbi_id:63, #linkacc:"M22270", #locus:"HUMAREPBS",
             #title:"Human alphoid repetitive DNA repeats 1' monomer,
                    clone alpha-T(1360) 14-12 III."),
            (#ncbi_id:61, #linkacc:"M22294", #locus:"HUMAREPCQ",
             #title:"Human alphoid repetitive DNA repeats 1' monomer,
                    clone alpha-T(1360) 22-7 III."),
            (#ncbi_id:47, #linkacc:"M81230", #locus:"HUMASATAB",
             #title:"Human alpha satellite DNA sequence.")})}),
...}
```

The performance, measured on a SuperSPARC Server, of our prototype on this example is reasonable. The first record of the output was displayed within seconds and the whole query was completed in 10 minutes by the wall clock. It should also be pointed out that it took us less than 5 minutes to compose and write down our example query — largely due to the fact that external databases and their tools can be freely combined in a compositional manner in CPL. Hence the entire process of composing the query and executing it was accomplished in 15 minutes.

1.6 Statement

This dissertation is drawn from several joint works with my colleagues. The theoretical chapters contain results from the papers of Buneman, Libkin, Naqvi, Subrahmanyam, Suciu, Tannen, and myself [29, 26, 28, 132, 133, 134, 204, 31]. The practical chapters contain material from the working notes that Hart and I wrote [89, 90, 91, 33]. Lest I forget to indicate their contributions in specific places later on, let me enumerate them now.

Section 1.1 contains many ideas which can mostly be attributed to Buneman, Tannen, Naqvi, and Subrahmanyam [28, 26]. Sections 2.1 and 2.2 have their roots in Tannen, Buneman, and Wong [29] and owe as much to Buneman and Tannen as to myself. Section 3.4 is taken from Libkin and Wong [133] and owes as much to Libkin as to myself. Section 3.3 is founded on the linear-order-lifting trick taught to me by Libkin [130]. Section 4.2 is a theorem which was first proved by Libkin [130]. That my proof of the finite-cofiniteness of k-multi-cycle queries in Section 4.3 applies verbatim to k-strict-binary-trees was first noticed by Libkin [134]. Finally, the idea in Chapter 5 of indicating the introduction of a new variable in CPL by a slash is due to Buneman.

Section 1.5 is taken from Hart and Wong [90]; the prose is mine but the example itself (as are all other examples of biological queries) is due to Hart. Chapter 9 is based on Hart and Wong [89, 91]; Hart deserves as much credit as I do in connecting Kleisli to so many biomedical systems. All C programs mentioned there, as well as part of the prose, are due to him. All ML programs mentioned there, as well as the design and the implementation of Kleisli, are due to me. Section 9.5 is based on an abstract of Buneman, Hart, and myself [33]; the vision presented there owes as much to Buneman and Hart as to myself.

Chapter 7 contains no new idea. It is included in this dissertation for the following three reasons:

- Several members on my proposal committee pressured me to consider the kind of optimizations mentioned there. The wisdom in this should be attributed to them.

- As the theory I am proposing is new, I think I should at least show that it does not hinder the application of known optimization techniques.
- The theme of my implementation in Chapter 9 is not the implementation of CPL; the real theme is extensibility or openness. Such a property is best demonstrated by showing how easy it is to extend the basic system. The classical operators and optimization rules are examples which most database practitioners are familiar with. Thus I use them for this purpose and so I present them in this unoriginal Chapter 7 in preparation for this ultimate purpose.

Chapter 8 is a collection of notes on experiments and thus contains nothing original. It is included in this dissertation for three reasons:

- To show that the prototype is working.
- To provide an idea of the performance of my prototype.
- To help illustrate the effect of the optimization rules of Chapters 6 and 7.

All remaining results, opinions, and *faults* in this dissertation are my own.

Part II

A logician's idle creations

Chapter 2

Querying Nested Relations

What is, was, or has been is not necessarily desirable. SIDNEY HOOK

When relational databases were introduced by Codd [41], a first-normal-form restriction was imposed on them. That is, the components of tuples in a relation were required to be atomic values. This constraint is considered unacceptable in many modern applications [141, 138, 102, 108, 42]. Subsequently, many nested relational databases were introduced.

The earliest of these was probably by Jaeschke and Schek [108] who allowed the components of tuples to be sets of atomic values. That is, nesting of relations was restricted to two levels. This restriction was relaxed by Thomas and Fischer [183], who allowed relations to be nested to arbitrary depth. Their algebraic query language consisted of the operators of flat relational algebra generalized to nested relations together with two operators for nesting and unnesting relations. However, their operators can only be applied to the outermost level of nested relations. Before a deeply nested relation could be manipulated, it was necessary to bring it up to the outermost level by a sequence of unnest operations; and after the manipulation, it was necessary to push the result back down to the right level of nesting by a sequence of nest operations. This constant need for restructuring was eliminated by Schek and Scholl [169], who introduced a recursive projection operator for navigation. The idea of recursive operator was taken further by Colby [45], who made all her operators

recursive. There were more complicated nested relational languages (such as Roth, Korth, and Silberschatz [168]; see also the comments of Tansel and Garnett [181]), which I prefer not to describe.

The design of nested relational query languages seems to be following a trend of increasing complexity. However, the increase in complexity is not always rewarded with an increase in expressive power. Specifically, the algebras of Thomas and Fischer [183], Schek and Scholl [169], and Colby [45] are all equivalent in expressive power. This complexity is an indication that some important simplifying concepts are lacking in the design of these languages. This chapter considers the use of $\mathcal{NRC}$, a calculus inspired by the categorical notion of a monad, as a nested relational language.

Organization

Section 2.1. A language based on restricting structural recursion on sets to $sru(\cup, f, \{\})$ is presented. This is the monad calculus initially proposed by Tannen, Buneman, and myself [29] and is referred to here as $\mathcal{NRC}$. $\mathcal{NRC}$ follows the programming language design principle of assigning to each fundamental type construction in the language a set of canonical operators and allowing these operators to be freely mixed. As a result, a full description of $\mathcal{NRC}$ can be presented in two pages.

Section 2.2. A fully algebraic version of $\mathcal{NRC}$, based on a more abstract presentation of monads, is given in this section. Functions definable in the algebra are shown to have polynomial time complexity. The equivalence between these two formulations are sketched. A large part of the proof is entirely equational. In contrast, the usual proof of equivalence between the relational algebra and the relational calculus is justified semantically. I use mainly $\mathcal{NRC}$ in this report, as it exhibits a good balance between abstractness and concreteness that is particularly suitable here. The algebraic version is more convenient for investigating the relationship between my languages and other nested relational algebras and I use it for this purpose in this chapter.

Section 2.3. Variants or tagged-unions are a useful data modeling concept [100] and are ubiquitous in modern programming languages [82]. I describe how they can be added to $\mathcal{NRC}$. The main result of this section is that variants do not change the expressive power of $\mathcal{NRC}$ in any essential way. A corollary of this result is that adding booleans and the conditional construct to $\mathcal{NRC}$ does not greatly affect its expressive power. The most interesting aspect of this result is its entirely equational proof. Few other nested relational query languages possess an equational theory strong enough for such a proof. This proof demonstrates the power of $\mathcal{NRC}$'s principled design over more ad hoc designs of many other nested relational languages.

Section 2.4. As it stands, $\mathcal{NRC}$ cannot express any non-monotonic operators such as the equality test. However, common non-monotonic operators such as the equality test, the membership test, the subset test, set intersection, set difference, and relational nesting are inter-definable using $\mathcal{NRC}$ as the ambient language. Since adding such operators to $\mathcal{NRC}$ does not take it out of polynomial time, this result strengthens a similar result of Gyssens and Van Gucht [84], who proved the inter-definability of these operators in the presence of the costly *powerset* operator. For this reason, this report uses the more convenient $\mathcal{NRC}(\mathbb{B},=)$ as its ambient language.

Section 2.5. As mentioned earlier, the nested relational languages of Thomas and Fischer [183], Schek and Scholl [169], and Colby [45] are equivalent in expressive power. I extend this result by proving that $\mathcal{NRC}(\mathbb{B},=)$ is also equivalent to these languages. Then I argue that $\mathcal{NRC}(\mathbb{B},=)$ can be profitably regarded as the "right" core for nested relational languages. I believe my results in this and in subsequent chapters are a convincing basis for this claim.

2.1 A query language based on the set monad

Structural recursion is a uniform paradigm for computing with collection types such as sets, bags, and lists. Tannen and Subrahmanyam [28] investigated the semantic aspect of structural recursion over sets, bags, and lists. They showed that certain preconditions must

be satisfied for structural recursion to be well defined. Tannen, Buneman, and Naqvi [26] demonstrated the connection of structural recursion to database query languages. They showed that by imposing suitable restrictions on structural recursion, a language equivalent to the traditional relational query language can be obtained. Tannen, Buneman, and I [29] restricted structural recursion in a different but more natural way that cuts structural recursion on sets down to homomorphisms over the set monoid (that is, the monoid with sets as objects, $\{\}$ as the identity, and $\cup$ as the binary operator).

The restricted form of structural recursion of Tannen, Buneman, and myself [29] results in an iteration mechanism on sets that corresponds to the central transformation on Kleisli triples [142]. There is a natural correspondence between Kleisli triples and monads [137]. Inspired by Moggi [146], Tannen, Buneman, and I [29] derived a calculus based on Kleisli triples and an algebra based on monads for querying nested relations. We showed, amongst other things, that the calculus and the algebra are equivalent. Inspired by the work of Wadler on monad comprehension [198], I presented [204] an equivalent language in the comprehension style.

In this section, I present the calculus, which I named $\mathcal{NRC}$. The algebra is presented in the next section. The presentation of the comprehension language is delayed until Chapter 5. The calculus exhibits a good balance between the abstract and the concrete and is particularly suitable for my work in all subsequent chapters. The algebra is more abstract and is especially handy for the remainder of this chapter which studies the relationship between my languages and existing nested relational algebras. The comprehension version is most convenient for writing programs and is used as the basis for CPL, a practical by-product of this dissertation.

The types in $\mathcal{NRC}$

are either complex object types or are function types $s \to t$ where s and t are complex object types. The complex object types are given by the grammar below.

$$s, t ::= b \mid unit \mid s \times t \mid \{s\}$$

The semantics of a complex object type is just a set of complex objects. An object of type $s \times t$ is a pair whose first component is an object of type s and whose second component is an object of type t. An object of type $\{s\}$ is a finite set whose elements are objects of type s. The type *unit* has precisely one object, which I denote (). There are also some unspecified base types b.

The expressions of $\mathcal{NRC}$

are given in Figure 2.1 together with their typing rules. The type superscripts are usually

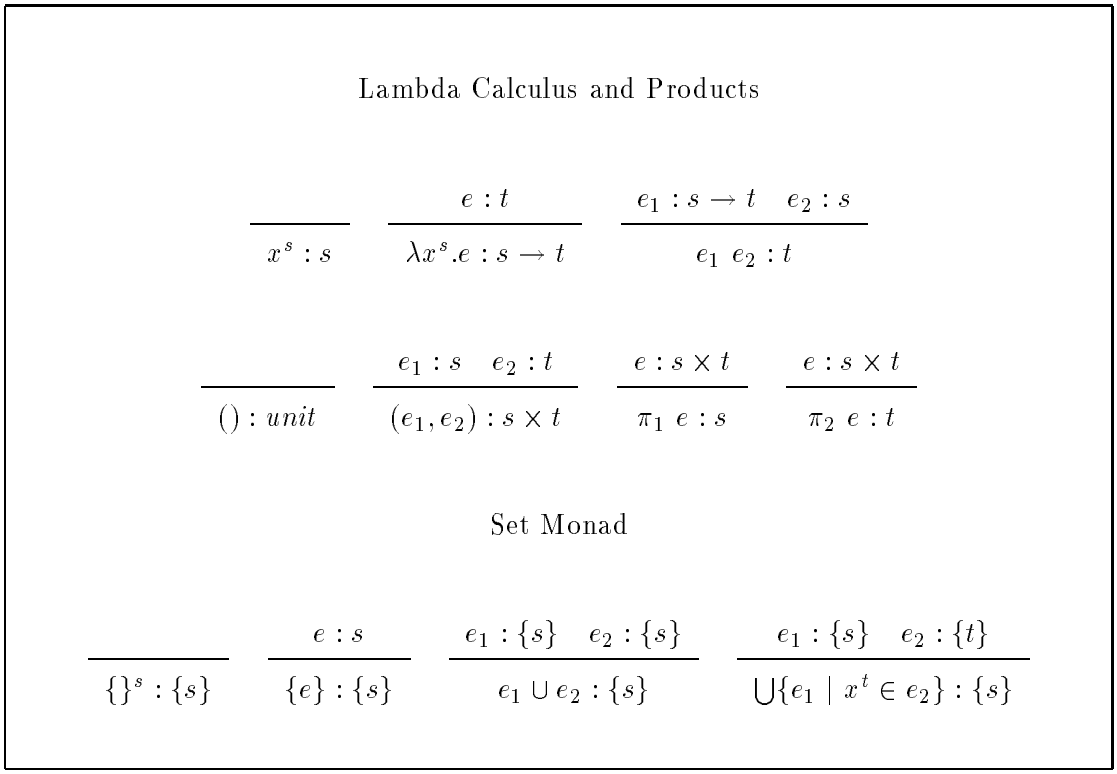


Figure 2.1: The expressions of $\mathcal{NRC}$.

omitted elsewhere in this report because they can be inferred [144]. (In fact, they remain inferrable even when records instead of pairs are used. See Ohori [154]; Ohori, Buneman, and Tannen [153]; Jategaonkar and Mitchell [110]; Remy [165]; etc.) The usual convention that bound variables are all distinct is adopted. Note also that I use the construct $\bigcup\{e_1 \mid x \in e_2\}$

instead of the equivalent $ext(\lambda x.e_1)(e_2)$ construct of Tannen, Buneman, and myself [29]. In later chapters, this basic language is extended by the introduction of new constants c of complex object type $Type(c)$ and new primitives p of function type $Type(p)$.

The semantics of these constructs are described below. The expression $\lambda x.e$ denotes the function f such that $f(x) = e$. The expression (e_1, e_2) denotes the pair whose first component is the object denoted by e_1 and whose second component is the object denoted by e_2. It has already been mentioned that () denotes the unique object of type *unit*. The expression $\pi_1\ e$ denotes the first component of the pair denoted by e. The expression $\pi_2\ e$ denotes the second component of the pair denoted by e. The expression $e_1\ e_2$ denotes the result of applying the function e_1 to the input e_2.

The expression $\{\}$ denotes the empty set. The expression $\{e\}$ denotes the singleton set containing the object denoted by e. The expression $e_1 \cup e_2$ denotes the union of the sets e_1 and e_2. The expression $\bigcup\{e_1 \mid x \in e_2\}$ denotes the set obtained by first applying the function $\lambda x.e_1$ to each object in the set e_2 and then taking their union; that is, $\bigcup\{e_1 \mid x \in e_2\} = f(o_1) \cup \ldots \cup f(o_n)$, where f is the function denoted by $\lambda x.e_1$ and $\{o_1, \ldots, o_n\}$ is the set denoted by e_2. In other words, $\bigcup\{e_1 \mid x \in e_2\}$ is really our restricted structural recursion $sru(\cup,\ \lambda x.e_1,\ \{\})(e_2)$.

Note that the $x \in e_2$ part in the $\bigcup\{e_1 \mid x \in e_2\}$ construct is not a membership test. It is an abstraction that introduces the variable x whose scope is the expression e_1. It should be understood in the same spirit in which the lambda abstraction $\lambda y.e$ is understood.

The $\bigcup\{e_1 \mid x \in e_2\}$ construct is the sole means in $\mathcal{NRC}$ for iterating over a set. More to the point, $\bigcup\{e_1 \mid x \in e_2\}$ is precisely the restricted form of structural recursion $sru(\cup, \lambda x.e_1, \{\})(e_2)$. It endows $\mathcal{NRC}$ with some basic capability for structural manipulations of nested relations. For example, the cartesian product of two sets X and Y can be defined as $cartprod(X, Y) \triangleq \bigcup\{\bigcup\{\{(x, y)\} \mid x \in X\} \mid y \in Y\}$. As a second example, the flattening of a nested set X can be defined as $flatten(X) \triangleq \bigcup\{x \mid x \in X\}$. As a last example, the projection of the first column of a relation X can be defined as $\Pi_1(X) \triangleq \bigcup\{\{\pi_1\ x\} \mid x \in X\}$.

The philosophy behind the design of $\mathcal{NRC}$ is very different from that of traditional query languages such as the flat relational algebra. The flat relational algebra is a rather ad hoc language and the only interesting thing about it is that it captures first-order logic. In contrast, the design of $\mathcal{NRC}$ follows very much in the spirit of Reynolds [167] and Cardelli [36]. Each distinct type construction in $\mathcal{NRC}$ is associated with a number of canonical assembly and dissembly operations that characterize the type construction in a universal sense: function abstraction and application for the arrow types; pair formation and projections for the tuple types; and set formations and iteration for the set types. The language is formed by allowing these constructs to be freely combined, provided typing constraints are satisfied. This philosophy on the design of modern programming languages can be seen in many books such as Gunter [82], Schmidt [170], etc. Indeed, one finds that the complaints of Codd [43, 44] and Date [53, 52] on the de facto query language SQL [106] cannot be applied to $\mathcal{NRC}$.

An equational theory for $\mathcal{NRC}$

The axioms for $\mathcal{NRC}$ are listed below. The reflexivity, symmetry, transitivity, congruence, and identities for $\{\}$ and $e_1 \cup e_2$ have been omitted. These axioms are the inspiration for the rewrite system used in Chapter 3 for proving the conservative extension property of $\mathcal{NRC}$ and in Chapter 6 for designing the pipelining rules in my optimizer. In the presentation of these rules, I write $e_1[e_2/x]$ for the expression obtained by replacing all free occurrences of the variable x in the expression e_1 by the expression e_2.

- $(\lambda x.e_1)(e_2) = e_1[e_2/x]$
- $\lambda x.e\ x = e$, if x is not free in e.
- $\pi_1(e_1, e_2) = e_1$
- $\pi_2(e_1, e_2) = e_2$
- $(\pi_1\ e,\ \pi_2\ e) = e$
- $e = ()$, if $e : \mathit{unit}$.
- $\bigcup\{e_1 \mid x \in \{e_2\}\} = e_1[e_2/x]$
- $\bigcup\{\{x\} \mid x \in e\} = e$
- $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} = \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$

These axioms are sound for $\mathcal{NRC}$. That is,

Proposition 2.1.1 *Let e_1 and e_2 be two $\mathcal{NRC}$ expressions. Suppose there is a proof of $e_1 = e_2$ using the axioms above. Then indeed $e_1 = e_2$ in our set-theoretic semantics.* □

In this dissertation, I use $e_1 = e_2$ to for all of the following situations: (1) e_1 and e_2 denote the same value, (2) a syntactic expression in my languages, and (3) the equality of e_1 and e_2 can be proved in the equational theories given in this dissertation. I rely on context to distinguish between the first sense and the second sense above. I always explicitly indicate the third sense as in Proposition 2.1.1 above. For simplicity, most of the soundness results are stated semantically, even though many parts of their proofs factor through the soundness result of Proposition 2.1.1 above.

2.2 Alternative 'monadic' formulation of the language

There is a natural correspondence between Kleisli triples and MacLane's monads; see Manes [142] for instance. While Kleisli triples correspond to the calculus $\mathcal{NRC}$, monads correspond to an algebra I name $\mathcal{NRA}$ here. This section presents the algebra, shows that it is polynomial-time bounded, and demonstrates its equivalence to $\mathcal{NRC}$.

The expressions of $\mathcal{NRA}$

are given in Figure 2.2 together with their typing rules. The meanings of these operators are as follow. The expression id is the identity function. The expression $g \circ h$ is the composition of functions g and h; that is, $(g \circ h)(x) = g(h(x))$. The expression ! is the terminator; hence $!(x) = ()$. The expressions π_1 and π_2 are respectively the first and the second projection on pairs. The expression $\langle g, h \rangle$ is pair formation; that is, $\langle g, h \rangle(x) = (g\ x,\ h\ x)$.

The expression η forms singleton set; that is, $\eta(x) = \{x\}$. The expression $K\{\}$ forms empty set; that is, $K\{\}() = \{\}$. The expression $\cup$ is the set union function. The expression μ flattens a set of sets; that is, $\mu\{X_1, \ldots, X_n\} = X_1 \cup \ldots \cup X_n$. The expression ρ_2 is the tensor function that pairs an object with every objects in a given set; that is,

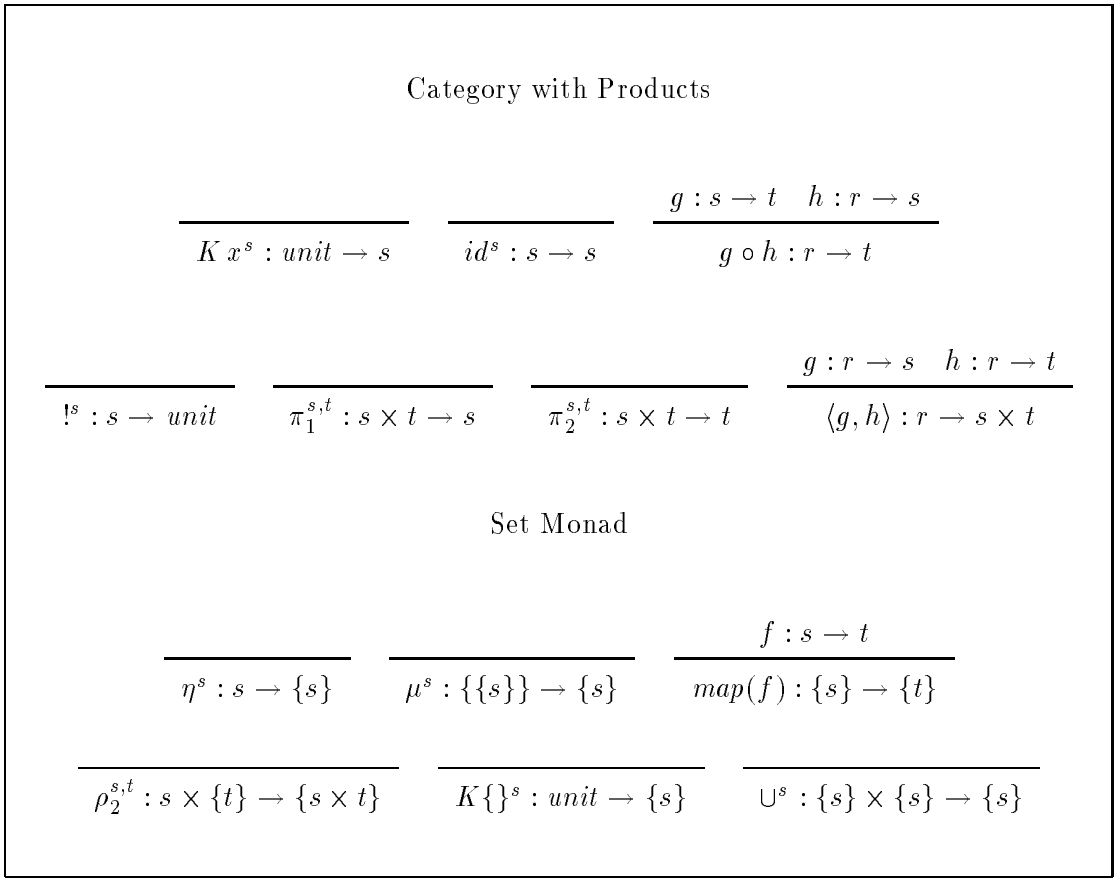


Figure 2.2: The expressions of $\mathcal{NRA}$.

$\rho_2(x, \{x_1, \ldots, x_n\}) = \{(x, x_1), \ldots, (x, x_n)\}$. The expression $map(f)$ is the function which applies f to every element in the input set; that is, $map(f)\{x_1, \ldots, x_n\} = \{f(x_1), \ldots, f(x_n)\}$.

The variables x of $\mathcal{NRC}$ correponds one-to-one to expressions Kx in $\mathcal{NRA}$. In addition, for each new primitive function $p : s \to t$ to be added to $\mathcal{NRC}$, p is added to $\mathcal{NRA}$ as an additional primitive. Also, for each new constant $c : s$ to be added to $\mathcal{NRC}$, a constant function $Kc : unit \to s$ is added to $\mathcal{NRA}$.

Note that all expressions in $\mathcal{NRA}$ have function types $s \to t$. Another interesting observation is that FQL [30], a language designed for the pragmatic purpose of communicating with network databases, was based roughly on the same set of operators as $\mathcal{NRA}$.

It is easy to see that for any reasonable definition of complex object size, $\mathcal{NRA}$ is always polynomial-time computable. A similar theorem can be proved for $\mathcal{NRC}$. In fact, a stronger version, where polynomiality under a specific operational semantics, can also be proved.

Theorem 2.2.1 *Let every additional primitive function p be computable in polynomial time with respect to the size of its input. Then every function definable in $\mathcal{NRA}$ is computable in polynomial time with respect to the size of its input.*

Proof. For any morphism expression f, a time-bound function $|f| : \mathbb{N} \to \mathbb{N}$ is given by

$$|f|(n) = \begin{cases} |g|(n) + |h|(n) & \text{if } f \text{ is } \langle g, h \rangle \\ |g|(|h|(n)) & \text{if } f \text{ is } g \circ h \\ n \times |g|(n) & \text{if } f \text{ is } map(g) \\ O(n^{k_p}) & \text{if } f \text{ is a primitive function } p, \text{ bound is by assumption} \\ O(n) & \text{otherwise} \end{cases}$$

□

An equational theory for $\mathcal{NRA}$

The axioms for $\mathcal{NRA}$ are listed below. The reflexivity, symmetry, transitivity, congruence, and identities for $K\{\}$ and $\cup$ have been omitted. In these axioms, I write $(f \times g)$ as a shorthand for $\langle f \circ \pi_1,\ g \circ \pi_2 \rangle$ and $\rightrightarrows$ as a shorthand for $\langle \pi_1 \circ \pi_1,\ \langle \pi_2 \circ \pi_1,\ \pi_2 \rangle \rangle$.

- $f \circ (g \circ h) = (f \circ g) \circ h$
- $f \circ id = f$
- $id \circ f = f$
- $\langle \pi_1 \circ f, \pi_2 \circ f \rangle = f$
- $\pi_1 \circ \langle f,\ g \rangle = f$
- $\pi_2 \circ \langle f,\ g \rangle = g$
- $f = !$, if $f : s \to unit$.
- $map(id) = id$
- $map(g \circ f) = map(g) \circ map(f)$
- $map(f) \circ \eta = \eta \circ f$
- $map(f) \circ \mu = \mu \circ map(map(f))$

- $id = \mu \circ \eta$
- $id = \mu \circ map(\eta)$
- $\mu \circ \mu = \mu \circ map(\mu)$
- $map(\pi_2) \circ \rho_2 = \pi_2$
- $\rho_2 \circ (id \times \eta) = \eta$
- $\rho_2 \circ (id \times \mu) = \mu \circ map(\rho_2) \circ \rho_2$
- $map(f \times g) \circ \rho_2 = \rho_2 \circ (f \times map(g))$
- $map(\rightrightarrows) \circ \rho_2 = \rho_2 \circ (id \times \rho_2) \circ \rightrightarrows$

These axioms are sound for $\mathcal{NRA}$. That is,

Proposition 2.2.2 *Let f and g be expressions of $\mathcal{NRA}$. Suppose there is a proof of $f = g$ according to the axioms above. Then indeed $f = g$ in our set-theoretic semantics.* □

The remainder of this section is devoted to working out the equivalence between $\mathcal{NRA}$ and $\mathcal{NRC}$.

Translating from $\mathcal{NRC}$ to $\mathcal{NRA}$

The following translation is due to Tannen and Buneman [29]. An expression $e : s$ of $\mathcal{NRC}$ is translated to an expression $\mathcal{A}[e] : unit \to s$ of $\mathcal{NRA}$, while an expression $e : s \to t$ of $\mathcal{NRC}$ is translated to an expression $\mathcal{A}[e] : s \to t$ of $\mathcal{NRA}$. In order to translate lambda abstraction, it is necessary to show that $\mathcal{NRA}$ enjoys a combinatorial completeness property [127]. Specifically, one can express abstraction of variables as a derived operation as follow. For any expression $h : s \to t$ of $\mathcal{NRA}$ and for any variable $x : r$, define an expression $\kappa x.h : r \times s \to t$ in $\mathcal{NRA}$ by

$$\begin{aligned}
\kappa x.h &\triangleq h \circ \pi_2 && \text{if } h \text{ does not contain } Kx \\
\kappa x.Kx &\triangleq \pi_1 \\
\kappa x.\langle f, g\rangle &\triangleq \langle \kappa x.f,\ \kappa x.g\rangle \\
\kappa x.(g \circ f) &\triangleq (\kappa x.g) \circ \langle \pi_1,\ \kappa x.f\rangle \\
\kappa x.map(f) &\triangleq map(\kappa x.f) \circ \rho_2
\end{aligned}$$

This operation satisfies the property that, in the equational theory of $\mathcal{NRA}$ above, there is a proof of $(\kappa x.h) \circ \langle Kx \circ !,\ id\rangle = h$. This property corresponds to the beta-conversion

rule of $\mathcal{NRC}$: $(\lambda x.e_1)(e_2) = e_1[e_2/x]$. With this, a description of the translation can now be given.

$$\frac{}{\mathcal{A}[()] \triangleq id : unit \to unit} \qquad \frac{\mathcal{A}[e_1] : unit \to s \quad \mathcal{A}[e_2] : unit \to t}{\mathcal{A}[(e_1, e_2)] \triangleq \langle \mathcal{A}[e_1], \mathcal{A}[e_2] \rangle : unit \to s \times t}$$

$$\frac{\mathcal{A}[e] : unit \to s \times t}{\mathcal{A}[\pi_1\ e] \triangleq \pi_1 \circ \mathcal{A}[e] : unit \to s} \qquad \frac{\mathcal{A}[e] : unit \to s \times t}{\mathcal{A}[\pi_2\ e] \triangleq \pi_2 \circ \mathcal{A}[e] : unit \to t}$$

$$\frac{\mathcal{A}[e] : unit \to t}{\mathcal{A}[\lambda x^s.e] \triangleq (\kappa x.\mathcal{A}[e]) \circ \langle id,\ !\rangle : s \to t} \qquad \frac{\mathcal{A}[e_1] : s \to t \quad \mathcal{A}[e_2] : unit \to s}{\mathcal{A}[e_1\ e_2] \triangleq\ \mathcal{A}[e_1] \circ \mathcal{A}[e_2] : unit \to t}$$

$$\frac{}{\mathcal{A}[x^s] \triangleq K\ x^s : unit \to s} \qquad \frac{}{\mathcal{A}[c] \triangleq K\ c : unit \to Type(c)}$$

$$\frac{}{\mathcal{A}[p] \triangleq p : Type(p)} \qquad \frac{\mathcal{A}[e] : unit \to s}{\mathcal{A}[\{e\}] \triangleq \eta\ \circ \mathcal{A}[e] : unit \to \{s\}}$$

$$\frac{\mathcal{A}[\lambda x.e_1] : t \to \{s\} \quad \mathcal{A}[e_2] : unit \to \{t\}}{\mathcal{A}[\bigcup\{e_1 \mid x \in e_2\}] \triangleq \mu \circ map(\mathcal{A}[\lambda x.e_1])\ \circ \mathcal{A}[e_2] : unit \to \{s\}}$$

$$\frac{}{\mathcal{A}[\{\}^s] \triangleq K\{\}^s : unit \to s} \qquad \frac{\mathcal{A}[e_1] : unit \to s \quad \mathcal{A}[e_2] : unit \to t}{\mathcal{A}[e_1 \cup e_2] \triangleq \cup \circ \langle \mathcal{A}[e_1], \mathcal{A}[e_2] \rangle}$$

Translating from $\mathcal{NRA}$ to $\mathcal{NRC}$

An expression $f : s \to t$ in $\mathcal{NRA}$ is translated to an expression $\mathcal{C}[f] : s \to t$ in $\mathcal{NRC}$. A description of the translation is given below.

$$\mathcal{C}[K\ x] \triangleq \lambda u.x \qquad \mathcal{C}[K\ c] \triangleq \lambda u.c \qquad \mathcal{C}[p] \triangleq p \qquad \mathcal{C}[id] \triangleq \lambda x.x$$

$$\mathcal{C}[\pi_1] \triangleq \lambda x.\pi_1\ x \qquad \mathcal{C}[\pi_2] \triangleq \lambda x.\pi_2\ x \qquad \mathcal{C}[!] \triangleq \lambda x.() \qquad \mathcal{C}[\eta] \triangleq \lambda x.\{x\}$$

$$\mathcal{C}[g \circ f] \triangleq \lambda x.\mathcal{C}[g](\mathcal{C}[f]\,x) \qquad \mathcal{C}[\langle f, g\rangle] \triangleq \lambda x.(\mathcal{C}[f]\,x, \mathcal{C}[g]\,x)$$

$$\mathcal{C}[\rho_2] \triangleq \lambda x.\bigcup\{\{(\pi_1\,x,\ y)\} \mid y \in\ \pi_2\,x\} \qquad \mathcal{C}[map(f)] \triangleq \lambda x.\bigcup\{\{\mathcal{C}[f]\,y\} \mid y \in x\}$$

$$\mathcal{C}[\mu] \triangleq \lambda x.\bigcup\{y \mid y \in x\} \qquad \mathcal{C}[\cup] \triangleq \lambda x.(\pi_1\ x) \cup (\pi_2\ x) \qquad \mathcal{C}[K\{\}] \triangleq \lambda x.\{\}$$

The equivalence of $\mathcal{NRA}$ and $\mathcal{NRC}$

There is an intimate connection between the equational theories $\mathcal{NRA}$ and $\mathcal{NRC}$. Namely, it can be shown that the translations preserve and reflect these theories. In fact, this result can be extended by adding arbitrary closed axioms. Similar results hold for the connection between simply typed lambda calculi and cartesian closed categories [27].

Theorem 2.2.3

- *Let f be an expression in $\mathcal{NRA}$. Then it can be proved in the theory of $\mathcal{NRA}$ that $\mathcal{A}[\mathcal{C}[f]] = f$.*
- *Let $e : s$ be an expression of $\mathcal{NRC}$ and x not free in e. Then it can be proved in the theory of $\mathcal{NRC}$ that $\mathcal{C}[\mathcal{A}[e]] = \lambda x.e$.*
- *Let $e : s \to t$ be an expression of $\mathcal{NRC}$. Then it can be proved in the theory of $\mathcal{NRC}$ that $\mathcal{C}[\mathcal{A}[e]] = e$.*
- *Let e_1 and e_2 be expressions of $\mathcal{NRC}$. Then it can be proved in the theory of $\mathcal{NRA}$ that $\mathcal{A}[e_1] = \mathcal{A}[e_2]$ if and only if it can be proved in the theory of $\mathcal{NRC}$ that $e_1 = e_2$.*
- *Let f and g be expressions of $\mathcal{NRA}$. Then it can be proved in the theory of $\mathcal{NRC}$ that $\mathcal{C}[f] = \mathcal{C}[g]$ if and only if it can be proved in the theory of $\mathcal{NRA}$ that $f = g$.* □

As a corollary of Proposition 2.1.1, Proposition 2.2.2 and Theorem 2.2.3, it is readily proved that the translations preserve semantics. Consequently, the equivalence between $\mathcal{NRA}$ and $\mathcal{NRC}$ is proved.

Corollary 2.2.4 *$\mathcal{NRA} = \mathcal{NRC}$ in the following sense:*

- *Let $f : s \to t$ be a closed expression in $\mathcal{NRA}$. Then $\mathcal{C}[f] = f$.*
- *Let $e : s \to t$ be a closed expression in $\mathcal{NRC}$. Then $\mathcal{A}[e] = e$.*
- *Let $e : s$ be a closed expression in $\mathcal{NRC}$. Then $\mathcal{A}[e] = \lambda x.e$, where $x : unit$ is arbitrary.*

Proof. The first item is proved by a routine structural induction on f. For the second item, let $e : s \to t$ be a closed expression in $\mathcal{NRC}$. By Theorem 2.2.3, $\mathcal{C}[\mathcal{A}[e]] = e$ is provable in the theory of $\mathcal{NRC}$. By Proposition 2.1.1, $\mathcal{C}[\mathcal{A}[e]] = e$. By the first item, we have $\mathcal{C}[\mathcal{A}[e]] = \mathcal{A}[e]$. Combining these two, we conclude $\mathcal{A}[e] = e$. The third item is similarly proved. □

An immediate benefit of the equivalence of the algebra and the calculus via translations is that constructs from both formalisms can be freely mixed. This combined language can be thought of as an extension by syntactic sugar of either the algebra or the calculus. It can also be regarded as a single formalism whose equational theory is obtained by joining the theories of $\mathcal{NRA}$ and $\mathcal{NRC}$ and adding the equations that define the translations between them. The result is a very rich and semantically sound theory.

2.3 Augmenting the language with variant types

Flat relations can be very inconvenient for certain applications. The need to encode complex information into flat format is sometimes an unnecessary hassle and can cause degradation in performance. Nested relations were introduced to alleviate the problem to some extent; see Makinouchi [141]. There remains some situations that are unnatural to model using nested relations. For example, how does one model the address of a person when it can be in very different formats such as his electronic mail identifier, his office address, or his home address?

In the Format data model of Hull and Yap [100] there is a type called the tagged-union. It is a good solution to the example problem. Intuitively, every object of a tagged-union carries a tag that indicates how it is injected into the union. Using it, one can define a contact address to be the tagged-union of electronic mail identifier, office address, and home address. Given such a contact address, its tag can be inspected to determine what kind of address it is before the appropriate processing is carried out. Such an idea was also present in the more complicated IFO data model of Abiteboul and Hull [4].

Tagged-unions correspond to variant types in programming languages [82] and to co-products in category theory [18]. A variant type $s + t$ is normally equipped with two assembly operations and one dissembly operation. One of the assembly operation is *left*; when it is applied to an object o of type s, it injects o into the variant type by tagging o with a left-tag. The other assembly operation is *right*; when it is applies to an object o of type t, it injects o into the variant type by tagging it with a right-tag. Note that if s and t are the same type, then for each object o in s, there is an object *left* o and an object *right* o in $s + t$ that correspond respectively to the left-tagged and right-tagged version of o. The dissembly is $(f \mid g)$; when it is applied to the left-tagged *left* o, it strips the tag and then applies f to o; when it is applied to the right-tagged *right* o, it strips the tag and then applies g to o.

This section adds variant types to $\mathcal{NRA}$ and to $\mathcal{NRC}$. Then I prove that the presence of variants contributes insignificantly to the expressive power of these languages. In fact, they add no expressive power when only functions of type $s \to \{t_1\} \times \ldots \times \{t_n\}$ are considered. For this reason variants are subsequently omitted from all my theoretical results on expressive power. However, being equally expressive does not mean being equally convenient. For this reason, I allow them to re-emerge in Chapter 5 in the design of the concrete query language. Also, a limited form of variants, in the guise of the *if-then-else* construct, is used as part of $\mathcal{NRC}$ for the same reason in all subsequent chapters.

Syntax and axioms for variants in $\mathcal{NRA}$

Let me write $\mathcal{NRA}^+$ for $\mathcal{NRA}$ extended with variants. The additional expressions for $\mathcal{NRA}^+$ are listed in Figure 2.3. The meaning of the top three constructs have already been explained in the introduction to this section. The δ_2 primitive prescribes the interaction between pairs and variants. It is the function such that $\delta_2(x, \mathit{left}\ y) = \mathit{left}(x, y)$ and $\delta_2(x, \mathit{right}\ y) = \mathit{right}(x, y)$.

The additional axioms for $\mathcal{NRA}^+$ are given below. The first three are the usual ones for variants. The remaining five axiomatize the distribution of pairs into variants.

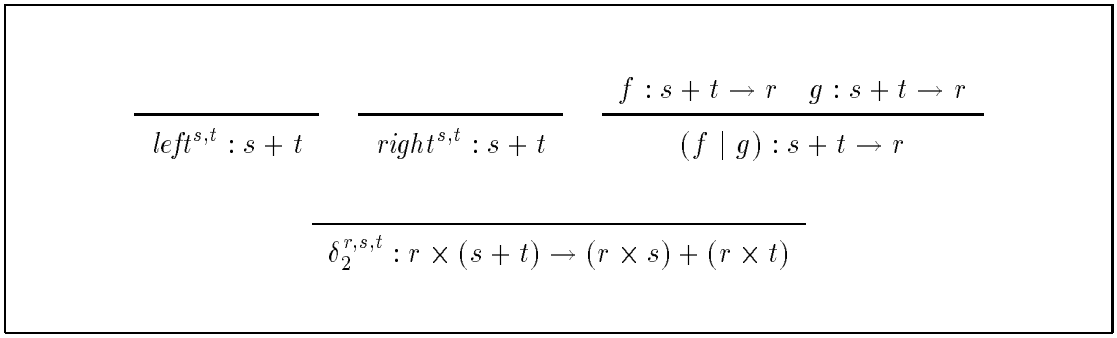


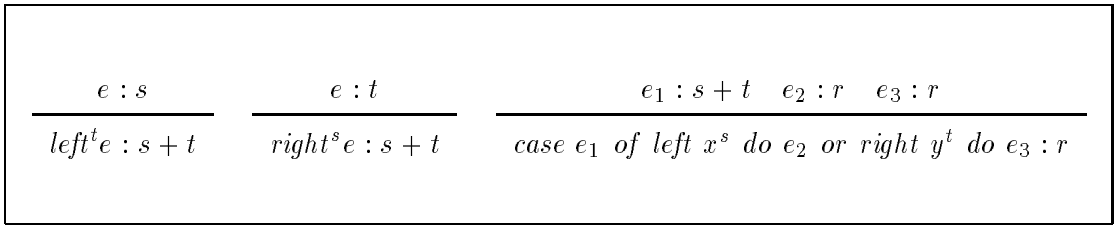
Figure 2.3: The variant constructs of $\mathcal{NRA}^+$.

- $(g \circ left \mid g \circ right) = g$
- $(g \mid f) \circ left = g$
- $(g \mid f) \circ right = f$
- $(left \circ \pi_2 \mid right \circ \pi_2) \circ \delta_2 = \pi_2$
- $(\pi_1 \mid \pi_1) \circ \delta_2 = \pi_1$
- $\delta_2 \circ (id \times left) = left$
- $\delta_2 \circ (id \times right) = right$
- $\delta_2 \circ (f \times (right \circ g \mid left \circ h)) = (right \circ (f \times g) \mid left \circ (f \times h)) \circ \delta_2$

Despite their apparent simplicity, these equations for $\mathcal{NRA}^+$ are sufficiently strong for proving a large number of identities. Let me list three here: $(f \circ g \mid f \circ h) = f \circ (g \mid h)$, $((f \times g) \mid (f \times h)) \circ \delta_2 = (f \times (g \mid h))$, and $\delta_2 \circ (f \times (left \circ g \mid right \circ h)) = (left \circ (f \times g) \mid right \circ (f \times h))$. The first one reveals that $(\cdot \circ \cdot)$ distributes over $(\cdot \mid \cdot)$. The second one shows the absorption of δ_2. The third one illustrates the naturality of δ_2.

Syntax and axioms for variants in $\mathcal{NRC}$

Let me write $\mathcal{NRC}^+$ for $\mathcal{NRC}$ extended with variants. The additional constructs of $\mathcal{NRC}^+$ are given in Figure 2.4. The *left* and *right* constructs have analogous meanings to the algebraic version. The semantics of *case* e_1 *of left* x *do* e_2 *or right* y *do* e_3 is as follow: if e_1 is an object *left* o, then the meaning of the whole expression is the object obtained by applying $\lambda x.e_2$ to o; if e_1 is an object *right* o, then the meaning of the whole expression is the object obtained by applying $\lambda x.e_3$ to o.



$$\frac{e : s}{left^t e : s + t} \qquad \frac{e : t}{right^s e : s + t} \qquad \frac{e_1 : s + t \quad e_2 : r \quad e_3 : r}{case\ e_1\ of\ left\ x^s\ do\ e_2\ or\ right\ y^t\ do\ e_3 : r}$$

Figure 2.4: The variant constructs of $\mathcal{NRC}^+$.

The additional axioms for $\mathcal{NRC}^+$ are listed below.

- $(case\ left\ e_1\ of\ left\ x\ do\ e_2\ or\ right\ y\ do\ e_3) = e_2[e_1/x]$
- $(case\ right\ e_1\ of\ left\ x\ do\ e_2\ or\ right\ y\ do\ e_3) = e_3[e_1/x]$
- $(case\ e_1\ of\ left\ x\ do\ e_2(left\ x)\ or\ right\ y\ do\ e_2(right\ y)) = e_2\ e_1$, if x and y are not free in e_2.
- $(case\ (case\ e\ of\ left\ x_1\ do\ e_1\ or\ right\ x_2\ do\ e_2)\ of\ left\ x_3\ do\ e_3\ or\ right\ x_4\ do\ e_4) = (case\ e\ of\ left\ x_1\ do\ (case\ e_1\ of\ left\ x_3\ do\ e_3\ or\ right\ x_4\ do\ e_4)\ or\ right\ x_2\ do\ (case\ e_2\ of\ left\ x_3\ do\ e_3\ or\ right\ x_4\ do\ e_4))$, if x_1 and x_2 not free in e_3 and e_4 and x_3 and x_4 not free in e_2.

These identities are the usual ones for variant types in lambda calculi. Let me provide two examples of the useful and interesting identities I have derived. The first one corresponds to a rule for migrating a piece of invariant code out of a loop: $\bigcup\{(case\ e_1\ of\ left\ x\ do\ e_2\ or\ right\ y\ do\ e_3) \mid z \in e_4\} = case\ e_1\ of\ left\ x\ do\ \bigcup\{e_2 \mid z \in e_4\}\ or\ right\ y\ do\ \bigcup\{\ e_3 \mid z \in e_4\}$, where x and y not free in e_4 and z not free in e_1. The second example is actually a kind of filter promotion: $(e)(case\ e_1\ of\ left\ x\ do\ e_2\ or\ right\ y\ do\ e_3) = case\ e_1\ of\ left\ x\ do\ e\ e_2\ or\ right\ y\ do\ e\ e_2$, if x and y not free in e.

Equivalence of $\mathcal{NRA}$ and $\mathcal{NRC}$ in the presence of variants

Now we need to extend the translations between $\mathcal{NRA}$ and $\mathcal{NRC}$ to deal with these new variant constructs. Three changes are requred.

First, we modify the definition for $\kappa x.h$, which translates the abstraction of variables. For the case when h is *left*, *right*, δ_2, or $(f \mid g)$ such that Kx does not occur in f and g, the existing definition can be used. Specifically, $\kappa x.h \triangleq h \circ \pi_2$. For the case of $(f \mid g)$ and Kx occurs in f or in g, then $\kappa.(f \mid g) \triangleq (\kappa x.f \mid \kappa x.g) \circ \delta_2$.

Second, we modify the definition of $\mathcal{A}[\cdot]$, which translates an expression of the calculus to an expression of the algebra: $\mathcal{A}[\mathit{left}\ e] \triangleq \mathit{left} \circ \mathcal{A}[e]$; $\mathcal{A}[\mathit{right}\ e] \triangleq \mathit{right} \circ \mathcal{A}[e]$; and $\mathcal{A}[\mathit{case}\ e_1\ \mathit{of}\ \mathit{left}\ x\ \mathit{do}\ e_2\ \mathit{or}\ \mathit{right}\ y\ \mathit{do}\ e_3] \triangleq (\mathcal{A}[\lambda x.e_2] \mid \mathcal{A}[\lambda y.e_3]) \circ \mathcal{A}[e_1]$.

Third, we modify the definition of $\mathcal{C}[\cdot]$, which translates a morphism of the algebra to an expression of the calculus: $\mathcal{C}[\mathit{left}] \triangleq \lambda x.\mathit{left}\ x$; $\mathcal{C}[\mathit{right}] \triangleq \lambda x.\mathit{right}\ x$; $\mathcal{C}[(f \mid g)] \triangleq \lambda x.\mathit{case}\ x\ \mathit{of}\ \mathit{left}\ y\ \mathit{do}\ \mathcal{C}[f](y)\ \mathit{or}\ \mathit{right}\ z\ \mathit{do}\ \mathcal{C}[g](z)$; and, $\mathcal{C}[\delta_2] \triangleq \lambda x.\mathit{case}\ \pi_2\ x\ \mathit{of}\ \mathit{left}\ y\ \mathit{do}\ \mathit{left}(\pi_1\ x,\ y)\ \mathit{or}\ \mathit{right}\ z\ \mathit{do}\ \mathit{right}(\pi_1\ x,\ z)$.

The extended translations have the desirable property of preserving and reflecting the equational theories of $\mathcal{NRA}^+$ and $\mathcal{NRC}^+$. In other words, a result analogous to Theorem 2.2.3 can be proved. Hence we conclude, in the same sense as Corollary 2.2.4,

Proposition 2.3.1 $\mathcal{NRA}^+ = \mathcal{NRC}^+$. □

Equivalence of $\mathcal{NRA}$ with and without variants

I now prove that $\mathcal{NRA}^+$ and $\mathcal{NRA}$ have the same expressive power. Instead of giving a semantic-based argument, a more interesting proof that is entirely equational is given. As a consequence, this argument works even when the set-theoretic semantics for our languages is replaced by some other kind of semantics. This proof requires several preliminary definitions. First extend $\mathcal{NRA}^+$ with an extra operator $\mathit{decollect}^s : \{s\} \to s$ that is required to

satisfy the equation: $decollect \circ \eta = id$. Note that *decollect* can be realized by any function that, on singleton input, returns the unique element in the input set. I denote $\mathcal{NRA}$ so extended with *decollect* by $\mathcal{NRA}(decollect)$. This convention of explicitly listing additional primitives is used throughout the dissertation.

Define s' by induction on s as follows:

- $b' \triangleq \{b\}$,
- $unit' \triangleq \{unit\}$,
- $(s \times t)' \triangleq \{s' \times t'\}$,
- $(s + t)' \triangleq \{(s' \times \{unit\}) + (t' \times \{unit\})\}$, and
- $\{s\}' \triangleq \{\{s'\}\}$.

Define $\varphi_s : s \to s'$ by induction on s as follows:

- $\varphi_{unit} \triangleq \eta$,
- $\varphi_b \triangleq \eta$,
- $\varphi_{s \times t} \triangleq \eta \circ (\varphi_s \times \varphi_t)$,
- $\varphi_{\{s\}} \triangleq \eta \circ map(\varphi_s)$, and
- $\varphi_{s+t} \triangleq \eta \circ (F \mid G)$, where $F \triangleq \langle\langle \varphi_s,\ \eta \circ !\rangle,\ \langle K\{\} \circ !,\ K\{\} \circ !\rangle\rangle$ and $G \triangleq \langle\langle K\{\} \circ !,\ K\{\} \circ !\rangle,\ \langle \varphi_t,\ \eta \circ !\rangle\rangle$.

Define $\varphi'_s : s' \to s$ by induction on s as follows:

- $\varphi'_{unit} \triangleq decollect$,
- $\varphi'_b \triangleq decollect$,
- $\varphi'_{s \times t} \triangleq (\varphi'_s \times \varphi'_t) \circ\ decollect$,

- $\varphi'_{\{s\}} \triangleq map(\varphi'_s) \circ decollect$, and
- $\varphi'_{s+t} \triangleq (left \circ \varphi'_s \circ \pi_1 \mid right \circ \varphi'_t \circ \pi_1) \circ decollect \circ \cup \circ (C \times D) \circ decollect$, where $C \triangleq \rho_2 \circ (left \times id)$ and $D \triangleq \rho_2 \circ (right \times id)$.

Essentially, φ_s and φ'_s form an encode-decode pair. The former encodes objects of type s, which may contain variants, into objects of type s', which contain no variants. The latter decodes the encoded objects to obtain the original objects. The encoding-decoding process is lossless.

Lemma 2.3.2 *There is a proof in the equational theory of $\mathcal{NRA}(decollect)$ that $\varphi'_s \circ \varphi_s = id$.* □

Let s be a type not involving variants. Define $\phi_s : s' \to \{s\}$ by induction on s as follows:

- $\phi_{unit} \triangleq id$,
- $\phi_b \triangleq id$,
- $\phi_{s \times t} \triangleq \mu \circ map(cartprod \circ (\phi_s \times \phi_t))$, where $cartprod \triangleq \mu \circ map(\rho_1) \circ \rho_2$, and
- $\phi_{\{s\}} \triangleq \eta \circ \mu \circ map(\mu \circ map(\phi_s))$.

Essentially, ϕ_s is a special decoding function for types which do not involve variants. Its most important property is that *decollect* does not occur in its definition. The encoding-decoding process is also lossless, except that the decoded result is placed in a singleton set.

Lemma 2.3.3 *Let s be a type not involving variants. Then there is a proof in the equational theory of $\mathcal{NRA}$ that $\phi_s \circ \varphi_s = \eta$.* □

Assume that for each unspecified primitive p in $\mathcal{NRA}^+$, $Type(p)$ does not involve variants. Then

Theorem 2.3.4 *For each expression $f : s \to t$ in $\mathcal{NRA}^+$, there is an expression $f' : s' \to t'$ in $\mathcal{NRA}$ such that the diagram below commutes in the theory of $\mathcal{NRA}^+$(decollect).*

$$\begin{array}{ccccc} s & \xrightarrow{f} & t & \xrightarrow{id} & t \\ {\scriptstyle\varphi_s}\downarrow & & {\scriptstyle\varphi_t}\downarrow & & \uparrow{\scriptstyle\varphi'_t} \\ s' & \xrightarrow[f']{} & t' & \xrightarrow[id]{} & t' \end{array}$$

Proof. The left square commutes by defining f' by induction on the structure of f as follow, where I write $\leftleftarrows$ as a shorthand for the inverse of $\rightrightarrows$:

- $Kc' \triangleq map(Kc)$
- $id' \triangleq id$
- $!' \triangleq map(!)$
- $\langle f, g\rangle' \triangleq \eta \circ \langle f', g'\rangle$
- $(f \circ g)' \triangleq f' \circ g'$
- $K\{\}' \triangleq map(K\{\})$
- $(map\ f)' \triangleq map(mapf')$
- $\eta' \triangleq \eta \circ \eta$
- $\mu' \triangleq map(\mu \circ map(\mu))$
- $\cup' \triangleq map(\cup) \circ map(\mu \times \mu)$
- $\rho_2' \triangleq map(\rho_2 \circ (id \times \mu))$
- $\pi_1' \triangleq \mu \circ map(\pi_1)$
- $\pi_2' \triangleq \mu \circ map(\pi_2)$
- $left' \triangleq \eta \circ \langle\langle id, \eta \circ !\rangle, \langle K\{\} \circ !, K\{\} \circ !\rangle\rangle$
- $right' \triangleq \eta \circ \langle\langle K\{\} \circ !, K\{\} \circ !\rangle, \langle id, \eta \circ !\rangle\rangle$
- $(f \mid g)' \triangleq \mu \circ map(\mu \circ \cup \circ ((map(\pi_1) \circ \rho_2 \circ (f' \times id)) \times (map(\pi_1) \circ \rho_2 \circ (g' \times id))))$
- $\delta' \triangleq map(\langle \Pi_1, \Pi_2\rangle \times \langle \Pi_1, \Pi_2\rangle) \circ map(\rho_2 \times \rho_2) \circ map(\leftleftarrows \times \leftleftarrows) \circ map\langle (id \times \pi_1), (id \times \pi_2)\rangle \circ \mu \circ map(\rho_2)$, where $\leftleftarrows \triangleq \langle \pi_1 \circ \pi_1, (\pi_2 \times id)\rangle$ and $\Pi_i \triangleq map(\pi_i)$.
- $p' \triangleq \mu \circ map(\varphi_t) \circ map(p) \circ \phi_s$, where $Type(p) = s \to t$.

The right square commutes by Lemma 2.3.2. Hence the theorem holds. □

As a consequence of Theorem 2.3.4, $\mathcal{NRA}^+$ and $\mathcal{NRA}$ have the same expressive power modulo the encoding and decoding functions φ and φ'. Hence in order to use $\mathcal{NRA}$ to "compute" an expression f definable in $\mathcal{NRA}^+$, the input and output must be appropriately encoded and decoded. If the type of f involves no variant types, such encoding and decoding

can be done away with.

Corollary 2.3.5 *$\mathcal{NRA} = \mathcal{NRA}^+$ in the following sense: Let f be an expression of $\mathcal{NRA}^+$. Let s and t be two types involving no variants. If $f : s \to t$, then there is an expression g of $\mathcal{NRA}$ such that there is a proof in the equational theory of $\mathcal{NRA}^+$ that $g = \eta \circ f$. If $f : s \to \{t\}$, then there is an expression h of $\mathcal{NRA}$ such that there is a proof in the equational theory of $\mathcal{NRA}^+$ that $f = h$.*

Proof. Define $g \triangleq \phi_t \circ f' \circ \varphi_s$. By Theorem 2.3.4, $g = \phi_t \circ \varphi_t \circ f$ is provable in $\mathcal{NRA}^+$. By Lemma 2.3.3, $g = \eta \circ f$ is provable in $\mathcal{NRA}^+$. The first item is thus proved. The second item follows immediately by defining $h \triangleq \mu \circ g$. □

Since we know that $\mathcal{NRA} = \mathcal{NRC}$ and that $\mathcal{NRA}^+ = \mathcal{NRC}^+$, Corollary 2.3.5 immediately gives us $\mathcal{NRC} = \mathcal{NRC}^+$ under the same conditions. That is, $\mathcal{NRC}$ is equivalent to $\mathcal{NRC}^+$ over the class of functions $f : s \to \{t\}$, where s and t involve no variants. This result is easily generalized to the class of functions $f : s \to \{t_1\} \times \ldots \times \{t_n\}$.

Let $\mathcal{NRC}$ extended with the usual boolean type $\mathbb{B}$ and associated constructs *true*, *false*, and *if-then-else* be denoted $\mathcal{NRC}(\mathbb{B})$. It is easy to see that these boolean constructs can be considered as a special case of variants as follow. Identify $\mathbb{B}$ with $unit + unit$. Identify *true* with *left*(). Identify *false* with *right*(). Identify *if* e_1 *then* e_2 *else* e_3 with *case* e_1 *of left* x *do* e_2 *or right* y *do* e_3. Therefore, we immediately obtain, in the same sense as Corollary 2.3.5,

Corollary 2.3.6 $\mathcal{NRC} = \mathcal{NRC}(\mathbb{B})$. □

2.4 Augmenting the language with equality tests

As it stands, $\mathcal{NRC}$ can perform many structural manipulations on nested relations. It is not yet adequate as a nested relational query language. In particular, it cannot express

any non-monotonic operations. (A monotonic operation, in the usual database sense, is an operation that preserves the inclusion ordering on sets.) To see this, let $\cap^s : \{s\} \times \{s\} \to \{s\}$ be the function that computes set intersection. Then

Proposition 2.4.1 *$\mathcal{NRC}$ cannot express $\cap^s$.*

Proof. Define an ordering $\sqsubseteq_s$ on complex objects of type s inductively:

- For base types: $o \sqsubseteq_b o$
- For pairs, pairwise ordering is used: $(o_1, o_2) \sqsubseteq_{s \times t} (o_1', o_2')$ if $o_1 \sqsubseteq_s o_1'$ and $o_2 \sqsubseteq_t o_2'$.
- For sets, the Hoare ordering is used: $O_1 \sqsubseteq_{\{s\}} O_2$ if for every $o_1 \in O_1$ there is some $o_2 \in O_2$ such that $o_1 \sqsubseteq_s o_2$.

Every function definable in $\mathcal{NRC}$ is monotone with respect to $\sqsubseteq$. However, $\cap^s$ is not. □

Therefore it is reasonable to add some extra primitives to $\mathcal{NRC}$. Booleans are not first-class citizens in popular relational query languages like the flat relational calculus and the flat relational algebra; see Maier [139], Ullman [189], etc. I stick with this tradition for now and simulate the booleans in $\mathcal{NRC}$ by treating the type $\{unit\}$ as the boolean type $\mathbb{B}$ and using $\{\}$ as *false* and $\{()\}$ as *true*. In this case, an equality test predicate on type s is a function $=^s$: $s \times s \to \{unit\}$ such that $(o = o') = \{\}$ whenever o and o' are distinct complex objects and $(o = o) = \{()\}$. The conditional can then be simulated as $(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \triangleq \bigcup\{e_2 \mid x \in e_1\} \cup \bigcup\{e_3 \mid x \in (e_1 = \{\})\}$. I write $\mathcal{NRC}(=)$ to denote $\mathcal{NRC}$ augmented with such an equality test at every type.

There are several other common non-monotonic operators commonly found in database query languages. Remarkably, it is not necessary to make ad hoc additions to $\mathcal{NRC}$ because all these operators are inter-definable when $\mathcal{NRC}$ is the ambient language.

Theorem 2.4.2 *The following languages are equivalent:*

- $\mathcal{NRC}(=)$, *where* $=^s : s \times s \to \mathbb{B}$ *is the equality test.*
- $\mathcal{NRC}(nest_2)$, *where* $nest_2^{s,t} : \{s \times t\} \to \{s \times \{t\}\}$ *is the relational nesting operator.*
- $\mathcal{NRC}(\cap)$, *where* $\cap^s : \{s\} \times \{s\} \to \{s\}$ *is set intersection.*
- $\mathcal{NRC}(\in)$, *where* $\in^s : s \times \{s\} \to \mathbb{B}$ *is the set membership test.*
- $\mathcal{NRC}(\subseteq)$, *where* $\subseteq^s : \{s\} \times \{s\} \to \mathbb{B}$ *is the subset inclusion test.*
- $\mathcal{NRC}(-)$, *where* $-^s : \{s\} \times \{s\} \to \{s\}$ *is set difference.*

Proof. Given $\subseteq^s$ one can define $\in^s$ as follow: $e_1 \in^s e_2 \triangleq \{e_1\} \subseteq^s e_2$. Given $\in^s$ one can define $=^s$ as follow: $e_1 =^s e_2 \triangleq \bigcup\{(e_2 \in^s \{e_1\}) \mid x \in (e_1 \in^s \{e_2\})\}$. Given $=^s$ one can define $\cap^s$ as follow: $e_1 \cap^s e_2 \triangleq \bigcup\{\bigcup\{\mathit{if}\ x =^s y\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\} \mid y \in e_2\} \mid x \in e_1\}$. Given both $=^{\{s\}}$ and $\cap^s$ one can define $\subseteq^s$ as follow: $e_1 \subseteq^s e_2 \triangleq (e_1 \cap^s e_2) =^{\{s\}} e_1$. Therefore, $\mathcal{NRC}(=) = \mathcal{NRC}(\cap) = \mathcal{NRC}(\in) = \mathcal{NRC}(\subseteq)$.

Given $=^s$ and $\in^s$ one can define $-^s$ as follow: $e_1 -^s e_2 \triangleq \bigcup\{\mathit{if}\ x \in^s e_2\ \mathit{then}\ \{\}\ \mathit{else}\ \{x\} \mid x \in e_1\}$. Given $-^s$ one can define $\cap^s$ as follow: $e_1 \cap^s e_2 \triangleq e_1 -^s (e_1 -^s e_2)$. Given $=^s$ one can define $nest_2^{s,t}$ as follow: $nest_2^{s,t}(e) \triangleq \bigcup\{\{(\pi_1(x),\ \bigcup\{\mathit{if}\ \pi_1(x) =^s \pi_1(y)\ \mathit{then}\ \{\pi_2(y)\}\ \mathit{else}\ \{\} \mid y \in e\})\} \mid x \in e\}$. Therefore, $\mathcal{NRC}(nest_2) \subseteq \mathcal{NRC}(=) = \mathcal{NRC}(\cap) = \mathcal{NRC}(\in) = \mathcal{NRC}(\subseteq) = \mathcal{NRC}(-)$.

I need to complete the cycle by deriving $-^s$ from $nest_2$. As this part of the proof is rather cunning, I use notations from both $\mathcal{NRC}$ and $\mathcal{NRA}$ to increase clarity. The operators from $\mathcal{NRA}$ appearing in this part of the proof are to be regarded as shorthands via the translation of Section 2.2.

Let $\leftrightarrows \triangleq \langle \pi_2,\ \pi_1 \rangle$. Let $\rho_1(e) \triangleq map(\leftrightarrows) \circ \rho_2 \circ \leftrightarrows$. Let $nest_1 \triangleq map(\leftrightarrows) \circ nest_2 \circ map(\leftrightarrows)$. Let $cartprod \triangleq \mu \circ map(\rho_1) \circ \rho_2$. To compute $R - S$, observe that $(map\langle\langle \pi_1, \lambda x.\{\}\rangle,\ \pi_2\rangle \circ nest_1 \circ nest_2 \circ \cup)(\rho_1(R, \{\}),\ \rho_1(S, \{\}))$ is a set containing possibly the following three pairs and nothing else:

$$U \triangleq \left\{ \begin{array}{lcll} (& (R \cap S,\ \{\}) & , & \{\{\},\ \{()\}\} &), \\ (& (R - S,\ \{\}) & , & \{\{\}\} &), \\ (& (S - R,\ \{\}) & , & \{\{()\}\} &) \end{array} \right\}$$

Now a way to select the second pair is needed. To accomplish this, let

$$W \triangleq \left\{ \begin{array}{lcll} (& (\{\},\ \{\}) & , & \{\{\},\ \{()\}\} &), \\ (& (\{\},\ \{()\}) & , & \{\{\}\} &), \\ (& (\{\},\ \{\}) & , & \{\{()\}\} &) \end{array} \right\}$$

Then $(map(\pi_1) \circ nest_1 \circ \cup)(U, W)$ produces a set consisting of three sets:

$$\left\{ \begin{array}{lcll} \{ & (R - S,\ \{\}) & , & (\{\},\ \{()\}) & \}, \\ \{ & (R \cap S,\ \{\}) & , & (\{\},\ \{\}) & \}, \\ \{ & (S - R,\ \{\}) & , & (\{\},\ \{\}) & \} \end{array} \right\}$$

This set is further manipulated to obtain the set consisting of the pairs below by applying the function $\mu \circ map(map(\pi_1 \times \pi_2)) \circ map(cartprod \circ \langle id,\ id \rangle)$:

$$V \triangleq \left\{ \begin{array}{lcll} (& R - S & , & \{\} &), \\ (& R - S & , & \{()\} &), \\ (& R \cap S & , & \{\} &), \\ (& \{\} & , & \{()\} &), \\ (& \{\} & , & \{\} &), \\ (& S - R & , & \{\} &) \end{array} \right\}$$

Using the fact that the product of any set with the empty set is empty, apply *cartprod* to each of these pairs to obtain the desired difference: $(map(\pi_1) \circ \mu \circ map(cartprod))(V)$. Thus the theorem is proved. □

A result similar to Theorem 2.4.2 was also proved by Gyssens and Van Gucht [84]. They showed that, in the presence of the *powerset* operator, those non-monotonic operators are inter-definable in the algebra of Schek and Scholl [169]. The algebra of Schek and Scholl is equivalent to $\mathcal{NRC}(=)$; see Theorem 2.5.4 below. In view of Theorem 2.2.1, the expensive

powerset operator is not definable in $\mathcal{NRC}(=)$. Consequently, Theorem 2.4.2 is a big improvement of Gyssens and Van Gucht's result. Incidentally, adding the *powerset* operator to $\mathcal{NRC}(=)$ gives us the nested relational algebra of Abiteboul and Beeri [2].

2.5 Equivalence to other nested relational languages

Other nested relational languages

The language of Thomas and Fischer [183] is the most widely known nested relational algebra. It consists of the five operators of the traditional flat relation algebra generalized to nested relations — namely: the relational projection operator that corresponds approximately to $\Pi_1 : \{s \times t\} \to \{s\}$, the relational selection operator that corresponds approximately to $select \triangleq \lambda X. \bigcup\{if\ \pi_1(x) = \pi_2(x)\ then\ \{x\}\ else\ \{\} \mid x \in X\}$, the join operator that corresponds approximately to $cartprod : \{s\} \times \{t\} \to \{s \times t\}$, the set union operator $\cup : \{s\} \times \{s\} \to \{s\}$, and the set difference operator $- : \{s\} \times \{s\} \to \{s\}$ — together with the relational nesting operator (that corresponds to $nest_2 : \{s \times t\} \to \{s \times \{t\}\}$) and the relational unnesting operator (that is roughly $flatten : \{\{s\}\} \to \{s\}$).

A major shortcoming of Thomas and Fischer's language is that all their operators must be applied to the top level of a relation. Therefore, to manipulate a relation X that is nested deeply inside another relation Y, it is necessary to first perform a sequence of unnesting operations to bring X up to the top level, then perform the manipulation, and finally perform a sequence of nesting operations to push the result back to where X was. These restructurings are inefficient and clumsy. The fact that the relational nesting and unnesting operators are not mutually inverse further compounds the problem.

The language of Schek and Scholl [169] is an extension of Thomas and Fischer's proposal. They parameterized the relational projection operator by a recursive scheme. The recursive scheme is specified in a language that mirrors their expression constructs, but it must be stressed that a scheme is not an expression. This projection operator gives them the ability

to navigate nested relations. The language of Colby [45] is an extension of the language of Schek and Scholl. Essentially, she parameterized all the rest of Thomas and Fischer's operators with a recursive scheme.

The unsatisfactory aspect in Schek and Scholl's (and also Colby's) proposal lies in the specification of the semantics of their language. The definition given by Schek and Scholl [169] for their projection operator contains more than 10 cases, one for each possible way of forming a scheme. This complicated semantics indicates a want of modularity in the design of their language. What seems to be missing here is the concept that functions can also be passed around and the concept of mapping a function over a set. As a result, Schek and Scholl [169] lamented their inability to provide their algebra with a useful equational theory.

Furthermore, the increased semantic complexity in the languages of Schek and Scholl [169] and of Colby [45] does not buy them any extra expressive power over the simple language of Thomas and Fischer [183].

Proposition 2.5.1 $Schek\&Scholl = Colby = Thomas\&Fischer.$

Proof. It is a theorem of Colby [45] that her algebra is expressible in Thomas and Fischer [183]. The latter is a sublanguage of Schek and Scholl [169], which is in turn a sublanguage of Colby's. □

This result of Colby is strengthened in this section by showing that my basic nested relational language $\mathcal{NRC}(=)$ coincides in expressive power with these three nested relational languages. Hence it can be argued that $\mathcal{NRC}(=)$ possesses just the right amount of expressive power for manipulating nested relations.

Description of $Thomas\&Fischer$

A detailed description of Thomas and Fischer's language is required for proving this result. Their language has types of the form $\{s_1 \times \ldots \times s_n \times s_{n+1}\}$. These types can obviously be

trivially encoded as types of the form $\{s_1 \times (\ldots \times (s_n \times s_{n+1})\ldots)\}$. Hence in my treatment below, I use only binary tuples.

- Union of sets. $\cup^s : \{s\} \times \{s\} \to \{s\}$. This one is already present in $\mathcal{NRC}(=)$.
- Intersection of sets. $\cap^s : \{s\} \times \{s\} \to \{s\}$. This one is definable in $\mathcal{NRC}(=)$ by Theorem 2.4.2.
- Set difference. $-^s : \{s\} \times \{s\} \to \{s\}$. This one is definable in $\mathcal{NRC}(=)$ by Theorem 2.4.2.
- Relational nesting. $nest_2^{s,t} : \{s \times t\} \to \{s \times \{t\}\}$. It is definable in $\mathcal{NRC}(=)$ by Theorem 2.4.2.
- Relational unnesting. $unnest_2^{s,t} : \{s \times \{t\}\} \to \{s \times t\}$. Its semantics can be defined in terms of $\mathcal{NRA}(=)$ as follow: $unnest_2 \triangleq \mu \circ map(\rho_2)$.
- Cartesian product. $cartprod^{s,t} : \{s\} \times \{t\} \to \{s \times t\}$. It is definable in $\mathcal{NRA}(=)$ as follow: $cartprod \triangleq \mu \circ map(\rho_1) \circ \rho_2$. The actual product operator used by Thomas and Fischer concatenates tuples. For example, it takes $\{s_1 \times s_2\} \times \{t_1 \times t_2\}$ to $\{s_1 \times (s_2 \times (t_1 \times t_2)\}$. But it is trivially decomposable into *cartprod*, which takes $\{s_1 \times s_2\} \times \{t_1 \times t_2\}$ to $\{(s_1 \times s_2) \times (t_1 \times t_2)\}$, followed by a projection operation to shift the brackets from $\{(s_1 \times s_2) \times (t_1 \times t_2)\}$ to $\{s_1 \times (s_2 \times (t_1 \times t_2)\}$.
- Their projection is the relational projection. That means it is a powerful operator that works on multiple columns; it can be used for making copies of any number of columns; and it can be used for permuting the positions of any number of columns. Such a powerful operator can be rarefied into five simpler operators: (1) the projection operator Π_2, which is equivalent to the function $map(\pi_2)$; (2) the shift-left operator $\Leftarrow^{r,s,t}$: $\{r \times (s \times t)\} \to \{(r \times s) \times t\}$, which is equivalent to the function $map(\leftleftarrows)$; (3) the shift-right operator $\Rightarrow^{r,s,t}$: $\{(r \times s) \times t\} \to \{r \times (s \times t)\}$, which is the inverse of the shift-left operator; (4) the switch operator $\Leftrightarrow^{s,t}$: $\{s \times t\} \to \{t \times s\}$, which is equivalent to the function $map(\leftrightarrows)$; and (5) the duplication operator $copy^s : \{s\} \to \{s \times s\}$, which is equivalent to $map\langle id, id\rangle$.

- Their selection operator is the relational selection and actually has the form $select(f, g)$ and can be interpreted as the function $\lambda x.\bigcup\{if\ f(y) = g(y)\ then\ \{y\}\ else\ \{\} \mid y\ \in x\}$. However, very severe restriction is placed on the form of f and g: they must be built entirely from π_1, π_2, $\langle\cdot,\cdot\rangle$, $\cdot \circ \cdot$, and id.

- As in the traditional relational algebra, Thomas and Fischer used letters to represent input relations. The letter R is reserved for this purpose and it is assumed to be distinct from all other variables. Finally, constant relations are written down directly. For example, $\{\{\}\}$ is the constant relation whose only element is the empty set. (Actually, the real McCoy did not have them. This absence of constants was an oversight of the original paper [183] and almost everyone assumed their presence; see Colby [45] for example. There were of course exceptions. For example, Van Gucht and Fischer [80] investigated normalization-lossless nested relations under the explicit assumption that constant relations, especially $\{\{\}\}$, were absent.)

A query is just an expression of complex object type such that R is its only free variable. Clearly every expression in the language of Thomas and Fischer can be treated as a shorthand of an expression in $\mathcal{NRC}(=)$. The rest of this section is devoted to proving the converse.

Equivalence of $\mathcal{NRC}(=)$ and $Thomas\&Fischer$

It is more convenient to prove this equivalence via a detour by restricting equality test to base types. Let $=^b: b \times b \to \mathbb{B}$ be the equality test on base types. Let $\neg : \mathbb{B} \to \mathbb{B}$ be boolean negation. (Note that $\mathbb{B}$ is really $\{unit\}$ at this point of the report. So $\neg\{\} = \{()\}$ and $\neg\{()\} = \{\}$.) I first show that $\mathcal{NRC}(=)$ and $\mathcal{NRC}(=^b, \neg)$ are equivalent. That is, equality test at every type can be expressed completely in terms of $=^b$ and $\neg$.

Lemma 2.5.2 $\mathcal{NRC}(=) = \mathcal{NRC}(=^b, \neg)$.

Proof. The right-to-left inclusion is obvious since $\neg e = (e =^{\mathbb{B}} \{\})$. For the left-to-right inclusion, it suffices to show that, with $\mathcal{NRC}$ as the ambient language, equality test $=^s$ at every type s can be defined in terms of equality test $=^b$ at base type and negation $\neg$.

Let us proceed by induction on s. For base types: $=^b$ is used. For pairs: $e_1 =^{s \times t} e_2 \triangleq$ *if* $\pi_1\ e_1 =^s \pi_1\ e_2$ *then* $\pi_2\ e_1 =^t \pi_2\ e_2$ *else* $\{\}$. For sets: $e_1 =^{\{s\}} e_2 \triangleq$ *if* $e_1 \subseteq^s e_2$ *then* $e_2 \subseteq^s e_1$ *else* $\{\}$. The subset test can be defined using negation as follow: $e_1 \subseteq^s e_2 \triangleq \neg \bigcup\{$*if* $x\ \in^s e_2$ *then* $\{\}$ *else* $\{()\} \mid x\ \in e_1\}$. The membership test can be defined as follow: $e_1 \in^s e_2 \triangleq \bigcup\{$*if* $x =^s e_1$ *then* $\{()\}$ *else* $\{\} \mid x\ \in e_2\}$. □

As a consequence, to prove the inclusion of $\mathcal{NRC}(=)$ in *Thomas&Fischer*, it suffices for us to prove the inclusion of $\mathcal{NRC}(=^b, \neg)$ in it instead. By Theorem 2.2.4, this inclusion reduces to the following:

Proposition 2.5.3 *$\mathcal{NRA}(=^b, \neg) \subseteq$ Thomas&Fischer over functions whose input-output are relations.*

Proof. Let $encode_s : s \to \{unit \times s\}$ be the function $encode_s(o) = \{((),\ o)\}$. Let $decode_t : \{unit \times t\} \to t$ be the partial function $decode_t\{((),\ o)\} = o$. Note that both $encode_s$ and $decode_t$ are definable in *Thomas&Fischer* when s and t are both products of set types. Suppose

Claim. For every closed expression $f : s \to t$ in $\mathcal{NRA}(=^b, \neg)$, for every complex object type r, there is an expression $f' : \{r \times s\} \to \{r \times t\}$ in *Thomas&Fischer* such that $f' = map(id \times f)$.

Then calculate as below:

•	$decode \circ f' \circ encode$	
=	$decode \circ map(id \times f) \circ encode$	By the claim above.
=	$decode \circ \eta\ \circ \langle !,\ f \rangle$	Definition of $encode$
=	f	Definition of $decode$

It remains to provide a proof of the claim. This proof is not difficult if one defines $f'(R)$ by induction on the structure of f as follows:

- $Kc'(R) \triangleq (\Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (R),\ \{c\}))))$
- $!'(R) \triangleq (\Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (R),\ \{()\}))))$
- $K\{\}'(R) \triangleq (\Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (R),\ \{\{\}\}))))$
- $\eta'(R) \triangleq \Pi_2(\Rightarrow (nest_2(\Leftarrow (copy(R)))))$
- $(g \circ f)'(R) \triangleq (g'(f'(R)))$
- $id'(R) \triangleq R$
- $\pi_1'(R) \triangleq \Longleftrightarrow (\Pi_2(\Rightarrow (\Longleftrightarrow (R))))$
- $\pi_2'(R) \triangleq \Longleftrightarrow (\Pi_1(\Leftarrow (\Longleftrightarrow (\Leftarrow (R)))))$
- $\langle f, g\rangle'(R) \triangleq \Rightarrow (\Pi_2(\Longleftrightarrow (\Rightarrow (\Rightarrow (\Pi_2(\Rightarrow (\Longleftrightarrow (\Pi_2(\Rightarrow (copy(\Rightarrow (\Longleftrightarrow (\Rightarrow (\Rightarrow (\Longleftrightarrow (\Pi_2(\Rightarrow (select(\pi_1 \circ \pi_1,\ \pi_1 \circ \pi_2)(cartprod(f'(\Longleftrightarrow (\Pi_2(\Rightarrow (copy(R))))), g'(\Longleftrightarrow (\Pi_2(\Rightarrow (copy(R))))))))))))))))))))))))))))))))$
- $(map\ f)'(R) \triangleq A(R) \cup\ B(R)$, where

  $A(R) \triangleq (\Longleftrightarrow (\Pi_1(\Leftarrow (\Longleftrightarrow (nest_2(f'(unnest_2(\Longleftrightarrow (\Pi_2(\Rightarrow (copy(R))))))))))))$,

  $B(R) \triangleq \Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{\}\}))))),\ \{\{\}\})))$.
- $\neg'(R) \triangleq A(R) \cup\ B(R)$, where

  $A(R) \triangleq \Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{\}\}))))),\ \{\{()\}\})))$,

  $B(R) \triangleq \Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{()\}\}))))),\ \{\{\}\})))$.
- $\rho_2'(R) \triangleq A(R) \cup\ B(R)$, where

  $A(R) \triangleq \Pi_2(\Rightarrow (nest_2(\Rightarrow (unnest_2(\Leftarrow (\Leftarrow (copy(R))))))))$,

  $B(R) \triangleq \Pi_2(\Rightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{\}\}))))),\ \{\{\}\})))$.

- $\mu'(R) \triangleq A(R) \cup\ B(R) \cup\ C(R)$, where

  $A(R) \triangleq nest_2(unnest_2(unnest_2(R)))$,

  $B(R) \triangleq \Pi_2(\Rrightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{\}\})))),\ \{\{\}\})))$,

  $C(R) \triangleq \Pi_2(\Rrightarrow (cartprod(\Longleftrightarrow (\Pi_1(select(\pi_2 \circ \pi_1,\ \pi_2)(cartprod(R,\ \{\{\{\}\}\})))),\ \{\{\}\})))$.

- $(=^b)'(R) \triangleq A(R) \cup\ B(R)$, where

  $A(R) \triangleq \Pi_2(\Rrightarrow (cartprod(\Longleftrightarrow (select(\pi_1 \circ \pi_2,\ \pi_2 \circ \pi_2)(R)),\ \{\{()\}\})))$,

  $B(R) \triangleq \Pi_2(\Rrightarrow (cartprod(\Longleftrightarrow (R - select(\pi_1 \circ \pi_2,\ \pi_2 \circ \pi_2)(R)),\ \{\{\}\})))$. □

Therefore, over relational input-output,

Theorem 2.5.4 $\mathcal{NRC}(=) = Thomas\&Fischer = Schek\&Scholl = Colby.$ □

As all these languages are equivalent in expressive power, one has to compare them in terms of some other characteristics. As it is inconvenient to write queries in the language of Thomas and Fischer, it is not a good candidate for the "right" nested relational language. As it is inconvenient to reason about queries in the languages of Schek and Scholl and of Colby, they are not good candidates either. So $\mathcal{NRC}(=)$ is a better candidate than them.

$\mathcal{NRC}(=)$ uses simulated booleans. However, it is more convenient to add the booleans as a base type $\mathbb{B}$ and the conditional directly to the language. I denote by $\mathcal{NRC}(\mathbb{B})$ the language obtained by augmenting $\mathcal{NRC}$ with the constructs in Figure 2.5.

$$\frac{}{true : \mathbb{B}} \qquad \frac{}{false : \mathbb{B}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{if\ e_1\ then\ e_2\ else\ e_3 : s}$$

Figure 2.5: The constructs for the Boolean type.

According to Corollary 2.3.6, this augmentation does not drastically modify $\mathcal{NRC}(=)$. I therefore strengthen my claim about $\mathcal{NRC}(=)$ to the following:

Claim 2.5.5 *$\mathcal{NRC}(\mathbb{B},=)$ is the "right" nested relational language.*

And from this point onwards, I use $\mathcal{NRC}(\mathbb{B},=)$ as the ambient language within which all subsequent results are developed.

Chapter 3

Conservative Extension Properties

The height of a complex object is the maximal depth of nesting of sets in the complex object. Suppose the class of functions, whose input has height at most i and output has height at most o, definable in a particular language is independent of the height of intermediate data used. Then that language is said to have the conservative extension property. This chapter proves that $\mathcal{NRC}(\mathbb{B},=)$ and several of its extensions possess the conservative extension property, which is then used to prove several interesting results.

Organization

Section 3.1. A strong normalization result is obtained for the nested relational language $\mathcal{NRC}(\mathbb{B},=)$. The induced normal form is then used to show that $\mathcal{NRC}(\mathbb{B},=)$ has the conservative extension property. The proof in fact holds uniformly across sets, bags, and lists, even in the presence of variant types. Paredaens and Van Gucht [159] proved a similar result for the special case when $i = o = 1$. Their result was complemented by Hull and Su [99] who demonstrated the failure of independence when the *powerset* operator is present and $i = o = 1$. The theorem of Hull and Su was generalized to all i and o by Grumbach and Vianu [79]. My result generalizes Paredaens and Van Gucht's to all i and o, providing a counterpart to the theorem of Grumbach and Vianu. A corollary of this result is that

$\mathcal{NRC}(\mathbb{B}, =)$, when restricted to flat relations, has the same power as the flat relational algebra [41].

Section 3.2. As a result $\mathcal{NRC}(\mathbb{B}, =)$ cannot implement some aggregate functions found in real database query languages such as the "select average from column" of SQL [106]. I therefore endow the basic nested relational language with rational numbers, some basic arithmetic operations, and a summation construct. The augmented language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is then shown to possess the conservative extension property. This result is new because conservativity in the presence of aggregate functions had never been studied before.

Section 3.3. $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is augmented with a linear order on base types. It is then shown that the linear order can be lifted within $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ to every complex object type. The augmented language also has the conservative extension property. This fact is then used to prove a number of surprising results. As mentioned earlier, Grumbach and Vianu [79] and Hull and Su [99] proved that the presence of *powerset* destroys conservativity in the basic nested relational language. My theorem shows that this failure can be repaired with very little extra machinery. Finite-cofiniteness results from the next chapter shows that this theorem does not follow from Immerman's [103] result on fixpoint queries in the presence of linear orders.

Section 3.4. A notion of internal generic family of functions is defined. It is then shown that the conservative extension property of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ endowed with (well-founded) linear orders can be preserved in the presence of any such family of functions. This result is a deeper explanation of the surprising conservativity of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ in the presence of *powerset* and other polymorphic functions.

3.1 Nested relational calculus has the conservative extension property at all input and output heights

Conservative extension property

The height $ht(s)$ of a type s is defined by induction on the structure of type; it is essentially the maximal depth of nesting of set-brackets in the type:

- $ht(unit) = ht(b) = 0$
- $ht(s \times t) = ht(s \to t) = \max(ht(s),\ ht(t))$
- $ht(\{s\}) = 1 + ht(s)$

Every expression of $\mathcal{NRC}$ has a unique typing derivation. The height of an expression e can thus be defined as $ht(e) = \max\{ht(s) \mid s \text{ occurs in the type derivation of } e\}$.

Definition 3.1.1 Let $\mathcal{L}_{i,o,k}$ be the class of functions definable by an expression $f : s \to t$ in the language $\mathcal{L}$, where $ht(s) \leq i$, $ht(t) \leq o$, and $ht(f) \leq k$. The language $\mathcal{L}$ is said to have the **conservative extension property** at input height i and output height o with displacement d and fixed constant c if $\mathcal{L}_{i,o,k} = \mathcal{L}_{i,o,k+1}$ for every $k \geq \max(i + d, o + d, c)$. □

My aim in this section is to show that $\mathcal{NRC}(\mathbb{B}, =)$ has the conservative extension property. Towards this end, consider the

Strongly normalizing rewrite system

consisting of the rules below.

- $(\lambda x.e_1)(e_2) \rightsquigarrow e_1[e_2/x]$
- $\pi_i(e_1,\ e_2) \rightsquigarrow e_i$

- $\bigcup\{e \mid x \in \{\}\} \rightsquigarrow \{\}$
- $\bigcup\{e_1 \mid x \in \{e_2\}\} \rightsquigarrow e_1[e_2/x]$
- $\bigcup\{e \mid x \in (e_1 \cup e_2)\} \rightsquigarrow \bigcup\{e \mid x \in e_1\} \ \cup \ \bigcup\{e \mid x \in e_2\}$
- $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \rightsquigarrow \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$
- $\bigcup\{e \mid x \in (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3)\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \bigcup\{e \mid x \in e_2\}\ \mathit{else}\ \bigcup\{e \mid x \in e_3\}$
- $\pi_i\ (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \pi_i\ e_2\ \mathit{else}\ \pi_i\ e_3$
- $\mathit{if}\ \mathit{true}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow e_2$
- $\mathit{if}\ \mathit{false}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow e_3$

These rules are derived from the theory of $\mathcal{NRC}^+$ by giving the axioms the orientation above. Clearly, they are sound. That is,

Proposition 3.1.2 *If $e_1 \rightsquigarrow e_2$, then $e_1 = e_2$.* □

A rewrite system is strongly normalizing if there is no infinite sequence of rewriting in that system. That is, after a finite number of rewrite steps, we must arrive at an expression to which no rewrite rule is applicable. The resulting expression is sometimes known as a normal form of the rewrite system.

Proposition 3.1.3 *The rewrite system induced by the rewrite rules above is strongly normalizing.*

Proof. Let φ maps variable names to natural numbers greater than 1. Let $\varphi[n/x]$ be the function that maps x to n and agrees with φ on other variables. Let $\|e\|\varphi$, defined below, measure the size of e in the environment φ where each free variable x in e is given the size $\varphi(x)$.

- $\|x\|\varphi = \varphi(x)$

- $\|true\|\varphi = \|false\|\varphi = \|c\|\varphi = \|()\|\varphi = \|\{\}\|\varphi = 2$
- $\|\pi_1\ e\|\varphi = \|\pi_2\ e\|\varphi = \|\{e\}\|\varphi = 2 \cdot \|e\|\varphi$
- $\|\lambda x.e\|\varphi = \|e\|\varphi[2/x]$
- $\|(\lambda x.e)(e')\|\varphi = \|e\|\varphi[\|e'\|\varphi/x] \cdot \|e'\|\varphi$
- $\|e_1\ \cup\ e_2\|\varphi = \|(e_1, e_2)\|\varphi = 1 + \|e_1\|\varphi + \|e_2\|\varphi$
- $\|\bigcup\{e' \mid x \in e\}\|\varphi = (\|e'\|\varphi[\|e\|\varphi/x] + 1) \cdot \|e\|\varphi$
- $\|\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\|\varphi = \|e_1\|\varphi \cdot (1 + \|e_2\|\varphi + \|e_3\|\varphi)$

Define $\varphi \leq \varphi'$ if $\varphi(x) \leq \varphi'(x)$ for all x. It is readily seen that $\|\cdot\|\varphi$ is monotonic in φ. Furthermore, it is readily verified that whenever $e \rightsquigarrow e'$, we have $\|e\|\varphi > \|e'\|\varphi$ for any choice of φ. Therefore, the rewrite system is strongly normalizing. □

Proof of conservative extension property

Thus, every expression of $\mathcal{NRC}(\mathbb{B})$ can be reduced to a very simple normal form. These normal forms exhibit an interesting property. Assuming no additional primitive p is present,

Theorem 3.1.4 *Let $e : s$ be an expression of $\mathcal{NRC}(\mathbb{B})$ in normal form. Then $ht(e) \leq \max(\{ht(s)\} \cup \{ht(t) \mid t$ is the type of a free variable in $e\})$.*

Proof. Let k be the maximum height of the free variables in e. Now proceed by induction on $e : s$.

- Case $e : s$ is x, $()$, c, *true*, *false*, or $\{\}$. Immediate.
- Case $e : s$ is $\{e'\}$. Immediate by hypothesis on e'.
- Case $e : s$ is $(e_1, e_2) : t_1 \times t_2$ or $e_1 \cup e_2 : \{t\}$. Immediate by hypothesis on e_1 and e_2.

- Case $e : s$ is $\lambda x.e' : r \to t$. By hypothesis, $ht(e') \leq \max(k, ht(t), ht(r))$. So $ht(e) = \max(ht(e'), ht(s)) \leq \max(k, ht(s))$.

- Case $e : s$ is $\pi_i e'$. By assumption e' is a normal form of the rewrite system. By a simple analysis on normal forms, it can be shown that e' must be a (possibly null) chain of projection on a variable. The case thus holds.

- Case $e : s$ is $\bigcup\{e_1 \mid x \in e_2\}$. Because e is in normal form, e_2 must be a chain of projections on a free variable. Hence $ht(e_2) \leq k$. So $ht(x) = ht(e_2) - 1 < k$. Then, by hypothesis, $ht(e_1) \leq \max(k, ht(x), ht(s))$. Then $ht(e) = \max(ht(s), ht(e_1), ht(e_2)) \leq \max(k, ht(s))$.

- Case $e : s$ is *if* e_1 *then* e_2 *else* e_3, where $e_1 : \mathbb{B}$, $e_2 : s$ and $e_3 : s$. Since e is a normal form, e_1 must be a chain of projections on a free variable. Hence $ht(e_1) \leq k$. By hypothesis, $ht(e_2) \leq \max(k, ht(s))$. Similarly, $ht(e_3) \leq \max(k, ht(s))$. Now $ht(e) = \max(ht(e_1), ht(e_2), ht(e_3)) \leq \max(k, ht(s))$. □

This theorem implies that $\mathcal{NRC}(\mathbb{B})$ has the conservative extension at all input and output types with displacement 0 and constant 0. Since equality at all types can be expressed in terms of equality at base types $=^b : b \times b \to \mathbb{B}$ and the emptiness test $\neg : \{unit\} \to \mathbb{B}$ with $\mathcal{NRC}(\mathbb{B})$ as the ambient language, it is straightforward to argue that our nested relational language has the conservative extension property.

Corollary 3.1.5 *$\mathcal{NRC}(\mathbb{B}, =)_{i,o,k} = \mathcal{NRC}(\mathbb{B}, =)_{i,o,k+1}$ for all i, o, $k \geq \max(i, o)$.*

Proof. Given any expression $e' : s$ in $\mathcal{NRC}(\mathbb{B}, =)$. First, replace all occurrences of $=$ in it by its definition in terms of $=^b$ and $\neg$. How $=$ is implemented in terms of $=^b$ and $\neg$ is unimportant. In particular, heights need not be preserved! This is because the new expression $e : s$ is an expression of $\mathcal{NRC}(\mathbb{B}, =^b, \neg)$. Theorem 3.1.4 yields the conservative extension theorem for $\mathcal{NRC}(\mathbb{B}, =^b, \neg)$ with fixed constant 1 because the emptiness test primitive has height 1. Now, if $e : s$ is such that $ht(s) = 0$ and all free variables have height 0, then in any normal form of e, any occurrence of $\neg$ must appear in a context of the form

$\neg(e_1 \cup \ldots \cup e_n)$ where each of e_i has the form $\{\}$ or the form $\{\cdot\}$. So the normal form can be adjusted as follow: if each of e_i is $\{\}$, then replace this subexpression with *true*; otherwise replace it with *false*. The resulting expression contains no $\neg$. Thus the fixed constant is reduced to 0 as desired. □

As remarked earlier, the above result implies height of input and output dictates the kind of functions that our languages can express. In particular, using intermediate expressions of greater height does not add expressive power. This property is in contrast to languages considered by Kuper and Vardi [124]; Abiteboul and Beeri [2]; Abiteboul, Beeri, Gyssens and Van Gucht [1]; Grumbach and Vianu [79]; and Hull and Su [99]. The kind of functions that can be expressed their languages is not characterized by the height of input and output and is sensitive to the height of intermediate operators. The principal difference between my languages and these languages is that the *powerset* operator is not expressible in my languages (see Theorem 2.2.1) but is expressible in those other languages. This indicates a non-trivial contribution to expresive power by the *powerset* operator.

This result has a practical signifcance. Some databases are designed to support nested sets up to a fixed depth of nesting. For example, Jaeschke and Schek [108] considered nonfirst-normal-form relations in which attribute domains are limited to powersets of simple domains (that is, databases whose height is at most 2). $\mathcal{NRC}(\mathbb{B},=)$ restricted to expression of height 2 is a natural query language for such a database. But knowing that $\mathcal{NRC}(\mathbb{B},=)$ is conservative at all heights, one can instead provide the user with the entire language $\mathcal{NRC}(\mathbb{B},=)$ as a more convenient query language for this database, so long as queries have input and output heights not exceeding 2.

Furthermore, as a special case, it is easy to show that the basic nested relational calculus, with input and output restricted to flat relations, is in fact conservative over flat relational algebra.

Proposition 3.1.6 *Every function definable in $\mathcal{NRC}(\mathbb{B},=)$ from flat relations to flat relations is also definable in the traditional flat relational algebra.*

Proof. The direct proof based on analysis of normal forms of the above rewrite system can be found in my paper [203]. For an indirect proof, recall that $\mathcal{NRC}(\mathbb{B},=) = Schek\&Scholl$. Then use the result of Paredaens and Van Gucht [159] that $Schek\&Scholl$ is conservative over the flat relational algebra. □

Comparison with Paredaens and Van Gucht's technique

The proposition above is the result first proved by Paredaens and Van Gucht [159]. The key to the proof of the conservative extension theorem is the use of normal form. The heart of Paredaens and Van Gucht's proof is also a kind of normal form result. However, the following main distinctions can be made between our results:

- The Paredaens and Van Gucht result is a conservative property with respect to flat relational algebra. This result implies $\mathcal{NRC}_{i,o,k} = \mathcal{NRC}_{i,o,k+1}$ for $i = o = 1$. My theorem generalizes this to any i and o.
- The normal form used by Paredaens and Van Gucht is a normal form of logic formulae and the intuition behind their proof is that of logical equivalence and quantifier elimination. In my case, the inspiration comes from a well-known optimization strategy (see Wadler's early papers [194, 195] on this subject). In plain terms, I have evaluated the query without looking at the input and managed to flatten the query sufficiently until all intermediate operators of higher heights are optimized out. This idea is summarized by the rewrite rule $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \rightsquigarrow \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$, which eliminates the intermediate collection built by $\bigcup\{e_2 \mid y \in e_3\}$.
- It should be pointed out that the syntax of $\mathcal{NRC}$ can be given slightly different interpretations. For example, it can be used as a query language for bags by interpreting $\{\}$ as the empty bag, $\{e\}$ as the singleton bag, and $\cup$ as the additive union for bags. It is also possible to use it to query lists by interpreting $\{\}$ as the empty list, $\{e\}$ as the singleton list, and $\cup$ as concatenation of lists. My theorem holds uniformly for these other interpretations of $\mathcal{NRC}$. The theorem also holds (see my paper [204]) in

the presence of variant types. It is not clear that the proof given by Paredaens and Van Gucht is applicable in such cases.

The fact that the basic nested relational language is conservative with respect to the flat relational algebra has several consequences: transitive closure, parity test, cardinality test, etc. cannot be expressed in $\mathcal{NRC}(=)$. This fact in turn implies that the language of Abiteboul and Beeri [2], which is equivalent to $\mathcal{NRC}(=)$ augmented with the *powerset* operator, must express things like transitive closure via an extremely expensive excursion through the *powerset* operator; see Suciu and Paredaens [179].

As pointed out, Paredaens and Van Gucht's result involved a certain amount of quantifier elimination. There are several other general results in logic that were proved using quantifier elimination; see Gaifman [70], Enderton [59], etc. The pipeline rule is related to quantifier elimination. It corresponds to eliminating quantifier in set theory as $\{e \mid \Delta_1 \wedge (\exists x.x \in \{e' \mid \Delta'\}) \wedge \Delta_2\} \rightsquigarrow \{e[e'/x] \mid \Delta_1 \wedge \Delta' \wedge \Delta_2[e'/x]\}$. It is interesting to observe that the logical notion of quantifier elimination corresponds to the physical notion of getting rid of intermediate data. Nevertheless, I stress again that the pipeline rule makes sense across lists, bags, and sets but quantifier elimination does not.

3.2 Aggregate functions preserve conservative extension properties

Real database query languages

are usually equipped with some aggregate functions. For example, the mean value in a column can be selected in SQL [106]. To handle queries such as totaling up a column and averaging a column, several primitives must be added to my basic nested relational calculus. In this section, I consider adding rational numbers $\mathbb{Q}$ and the constructs depicted in Figure 3.1 to $\mathcal{NRC}(\mathbb{B}, =)$.

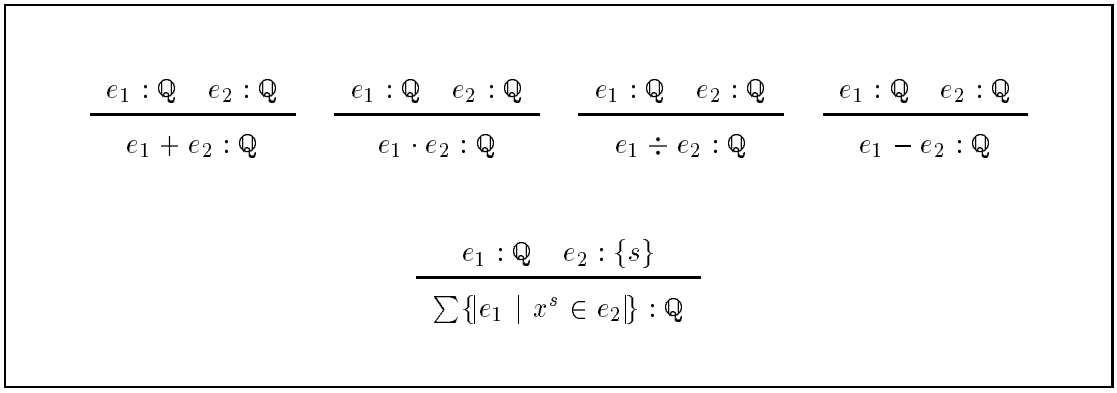


Figure 3.1: Arithmetic and summation operators for rational numbers.

The operators $+$, $\cdot$, $-$, and $\div$ are respectively addition, multiplication, subtraction, and division of rational numbers. The summation construct $\sum\{\!|e_1 \mid x^s \in e_2|\!\}$ denotes the rational obtained by first applying the function $\lambda x.e_1$ to every item in the set e_2 and then adding up the results. Hence $\sum\{\!|e_1 \mid x^s \in e_2|\!\} = f(o_1) + \ldots + f(o_n)$, where f is the function denoted by $\lambda x.e_1$ and $\{o_1, \ldots, o_n\}$ is the set denoted by e_2.

The extended language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is capable of expressing many aggregate operations found in practical databases. For instance, counting the number of records in R is $count(R) \triangleq \sum\{\!|1 \mid x \in R|\!\}$ and totaling up the first column of R is $total(R) \triangleq \sum\{\!|\pi_1\ x \mid x \in R|\!\}$. Another example is to take the average of the first column of R by $average(R) \triangleq total(R) \div count(R)$. A more sophisticated example is to calculate the variance of the first column of R as $variance(R) \triangleq (\sum\{\!|sq(\pi_1\ x) \mid x \in R|\!\} - (sq(\sum\{\!|\pi_1\ x \mid x \in R|\!\}) \div count(R))) \div count(R)$, where $sq(x) \triangleq x \cdot x$.

Aggregate functions were first introduced into flat relational algebra by Klug [119]. He introduced these functions by repeating them for every column of a relation. That is, $aggregate_1$ is for column 1, $aggregate_2$ is for column 2, and so on. Ozsoyoglu, Ozsoyoglu, and Matos [157] generalized this approach to nested relations. The summation construct is more general. On the other hand, Klausner and Goodman [117] had "stand-alone" aggregate functions such as $mean : \{\mathbb{Q}\} \to \mathbb{Q}$. However, they had to rely on a notion of hiding to deal correctly with duplicates. Hiding is different from projection. Let $R \triangleq \{(1,2),(2,3),(2,4)\}$.

Projecting out the second column of R gives us $R' \triangleq \{1, 2\}$. Hiding the second column of R gives us $R'' \triangleq \{(1, [2]), (2, [3]), (2, [4])\}$, where the hidden components are indicated by square brackets. Observe that the former "eliminates" duplicates as sets have no duplicate by definition. The latter "retains" the duplicated 2 by virtue of tagging them with different hidden components. Then $mean(R'')$ produces the average of the first column of R, whereas $mean(R')$ does not compute the mean correctly. The use of hiding to retain duplicates is rather clumsy. The summation construct is simpler.

In the remainder of this section, I show that $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ has the conservative extension property. The proof is a generalization of the previous proof. However, let me first replace $=^s$ and $\in^s$ with the syntactic sugars defined in the proposition below. It is important to observe that the $\bigcup\{e_1 \mid x \in e_2\}$ construct is not used in these syntactic sugars. This observation is crucial in verifying the claims on the measures $\|e\|\phi\theta$ and $\|e\|\theta$ used in the proof of Proposition 3.2.3.

Proposition 3.2.1 *Any equality test* $=^s : s \times s \to \mathbb{B}$ *can be implemented in terms of equality tests at base types* $=^b : b \times b \to \mathbb{B}$, *using* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ *as the ambient language.*

Proof. Proceed by induction on s.

- $=^b$ is the given equality test at base type b.
- $x =^{s \times t} y \triangleq$ *if* $\pi_1\ x =^s \pi_1\ y$ *then* $\pi_2\ x =^t \pi_2\ y$ *else false*
- $X =^{\{s\}} Y \triangleq$ *if* $X \subseteq^s Y$ *then* $Y \subseteq^s X$ *else false*, where
- $X \subseteq^s Y \triangleq ((\sum\{\!|$ *if* $x \in^s Y$ *then* 0 *else* $1 \mid x \in X|\!\}) =^{\mathbb{Q}} 0)$
- $x \in^s Y \triangleq (\sum\{\!|$ *if* $x =^s y$ *then* 1 *else* $0 \mid y \in Y|\!\}) =^{\mathbb{Q}} 1.$ □

More rewrite rules

Now, consider appending the rules below to those of the previous section.

- $\sum\{\!|e \mid x \in \{\}|\!\} \rightsquigarrow 0$
- $\sum\{\!|e \mid x \in \{e'\}|\!\} \rightsquigarrow e[e'/x]$
- $\sum\{\!|e \mid x \in \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3|\!\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \sum\{\!|e \mid x \in e_2|\!\}\ \mathit{else}\ \sum\{\!|e \mid x \in e_3|\!\}$
- $\sum\{\!|e \mid x \in e_1 \cup e_2|\!\} \rightsquigarrow \sum\{\!|e \mid x \in e_1|\!\} + \sum\{\!|\mathit{if}\ x \in e_1\ \mathit{then}\ 0\ \mathit{else}\ e \mid x \in e_2|\!\}$
- $\sum\{\!|e \mid x \in \bigcup\{e_1 \mid y \in e_2\}|\!\}$
  $\rightsquigarrow \sum\{\!|\sum\{\!|(e \div \sum\{\!|\sum\{\!|\mathit{if}\ x = v\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid v \in e_1|\!\} \mid y \in e_2|\!\}) \mid x \in e_1|\!\} \mid y \in e_2|\!\}$

This system of rewrite rules preserves the meanings of expressions. The last rule deserves special attention. Consider the incorrect equation: $\sum\{\!|e \mid x \in \bigcup\{e_1 \mid y \in e_2\}|\!\} = \sum\{\!|\sum\{\!|e \mid x \in e_1|\!\} \mid y \in e_2|\!\}$. Suppose e_2 evaluates to a set of two distinct objects $\{o_1, o_2\}$. Suppose $e_1[o_1/y]$ and $e_1[o_2/y]$ both evaluate to $\{o_3\}$. Suppose $e[o_3/x]$ evaluates to 1. Then the left-hand-side of the "equation" returns 1 but the right-hand-side yields 2. The division operation in the last rule is used to handle duplicates properly.

Proposition 3.2.2 *If $e_1 \rightsquigarrow e_2$, then $e_1 = e_2$.* □

While the last two rules seem to increase the "character count" of expressions, it should be remarked that $\sum\{\!|e_1 \mid x \in e_2|\!\}$ is always rewritten by these two rules to an expression that decreases in the e_2 position. This observation is the key to the following result:

Proposition 3.2.3 *The rewrite system above is strongly normalizing.*

Proof. Modify the measure $\|\cdot\|\varphi$ used in Proposition 3.1.3 to include $\|\sum\{\!|e' \mid x \in e|\!\}\|\varphi = (\|e'\|\varphi[\|e\|\varphi/x] + 1) \cdot \|e\|\varphi$. Then $\|e\|\varphi$ is monotone in φ. Moreover, if $e_1 \rightsquigarrow e_2$ via any rule but not the last two, then $\|e_1\|\varphi > \|e_2\|\varphi$. That is, this measure strictly descreases with respect to all the rules except the last two.

Let θ be a function that maps variable to natural numbers greater than 1. Let $\theta[n/x]$ be the function that maps x to n and agrees with θ on other variables. Let $\|e\|\theta$ be defined as below. Then $\|e\|\theta$ is monotone in θ. Moreover, if $e_1 \rightsquigarrow e_2$, then $\|e_1\|\theta \geq \|e_2\|\theta$.

- $\|true\|\theta = \|false\|\theta = \|c\|\theta = \|()\|\theta = \|\{\}\|\theta = 2$
- $\|\pi_1\ e\|\theta = \|\pi_2\ e\|\theta = \|e\|\theta$
- $\|x\|\theta = \theta(x)$
- $\|\lambda x.e\|\theta = \|e\|\theta[2/x]$
- $\|(\lambda x.e_1)(e_2)\|\theta = \max(\|e_2\|\theta, \|e_1\|\theta[\|e_2\|\theta/x])$
- $\|if\ e_1\ then\ e_2\ else\ e_3\|\theta = \max(\|e_1\|\theta, \|e_2\|\theta, \|e_3\|\theta)$
- $\|\{e\}\|\theta = 1 + \|e\|\theta$
- $\|e_1 \cup e_2\|\theta = 1 + \max(\|e_1\|\theta, \|e_2\|\theta)$
- $\|\bigcup\{e_1 \mid x \in e_2\}\|\theta = (\|e_1\|\theta[\|e_2\|\theta/x])^{\|e_2\|\theta}$
- $\|(e_1, e_2)\|\theta = \|e_1 + e_2\|\theta = \|e_1 - e_2\|\theta = \|e_1 \cdot e_2\|\theta = \|e_1 \div e_2\|\theta = \max(\|e_1\|\theta, \|e_2\|\theta)$
- $\|\sum\{|e_1 \mid x \in e_2\}\|\theta = \max(\|e_2\|\theta, \|e_1\|\theta[\|e_2\|\theta/x])$

Let σ denote an infinite tuple $(\ldots, \sigma(1), \sigma(0))$ with finitely many non-zero components. Let $\sigma_1 * \sigma_2$ denotes the tuple σ obtained by component-wise summation of σ_1 and σ_2. Let $\sigma[n]$ denotes the tuple σ' such that $\sigma'(n) = \sigma(n) + 1$ and $\sigma'(m) = \sigma(m)$ for $m \neq n$. Let ϕ be a function mapping variables to tuples σ's. Let $\phi[\sigma/x]$ maps x to the tuple σ and agrees with ϕ on other variables. Let $\|e\|\phi\theta$ be defined as below. Then $\|e\|\phi\theta$ is monotone in both ϕ and θ. Furthermore, if $e_1 \rightsquigarrow e_2$, then $\|e_1\|\phi\theta \geq \|e_2\|\phi\theta$. More importantly, if $e_1 \rightsquigarrow e_2$ via the last two rewrite rules above, then $\|e_1\|\phi\theta > \|e_2\|\phi\theta$. Thus this measure strictly decreases for the last two rules and remains unchanged for the other rules.

- $\|x\|\phi\theta = \phi(x)$
- $\|\lambda x.e\|\phi\theta = \|e\|\phi[(\ldots, 0)/x]\theta[2/x]$
- $\|(\lambda x.e_1)(e_2)\|\phi\theta = \|\bigcup\{e_1 \mid x \in e_2\}\|\phi\theta = \|e_2\|\phi\theta * \|e_1\|(\phi[\|e_2\|\phi\theta/x])(\theta[\|e_2\|\theta/x])$
- $\|true\|\phi\theta = \|false\|\phi\theta = \|c\|\phi\theta = \|()\|\phi\theta = \|\{\}\|\phi\theta = (\ldots, 0)$

- $\|\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\|\phi\theta = \|e_1\|\phi\theta * \|e_2\|\phi\theta * \|e_3\|\phi\theta$
- $\|\pi_1\ e\|\phi\theta = \|\pi_2\ e\|\phi\theta = \|\{e\}\|\phi\theta = \|e\|\phi\theta = \|e\|\phi\theta$
- $\|e_1 \cup e_2\|\phi\theta = \|e_1\|\phi\theta * \|e_2\|\phi\theta$
- $\|\sum\{|e_1 \mid x \in e_2|\}\|\phi\theta = (\|e_2\|\phi\theta * \|e_1\|\phi[\|e_2\|\phi\theta/x]\theta[\|e_2\|\theta/x])[\|e_2\|\theta]$

The termination measure for the rewrite system above can now be defined as $\|e\|\varphi\phi\theta = (\|e\|\phi\theta, \|e\|\varphi)$. Then $\|e\|\varphi\phi\theta$ is monotone in all of φ, ϕ, and θ. Furthermore, if $e_1 \rightsquigarrow e_2$, then $\|e_1\|\varphi\phi\theta > \|e_2\|\varphi\phi\theta$. Therefore, the rewrite system above is strongly normalizing. □

Conservative extension in the presence of aggregate functions

Finally, by a routine application of structural induction, we obtain the conservative extension property for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$.

Theorem 3.2.4 *Let $e : s$ be an expression of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ in normal form. Then $ht(e) \leq \max(\{ht(s)\} \cup \{ht(t) \mid t$ is the type of a free variable occurring in $e\})$. So $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ has the conservative extension property with fixed constant* 0. □

Conservativity in the presence of aggregate functions was not studied by earlier researchers. The theorem above implies that $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)_{i,o,h} = \mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)_{i,o,h+1}$ for any i, o, $h \geq \max(i, o)$. Hence I have generalized the result of Paredaens and Van Gucht [159] and my earlier theorem to the case where aggregate functions are present.

3.3 Linear ordering makes proofs of conservative extension properties uniform

The conservative extension property can be used to study many properties of languages (see Libkin and myself [130] for some examples). In Corollary 4.1.2, I use it to show that $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is incapable of expressing the usual linear ordering $\leq^{\mathbb{Q}}: \mathbb{Q} \times \mathbb{Q} \to \mathbb{B}$ on rational numbers. So I propose to augment $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ with a linear order $\leq^b: b \times b \to b$ for each base type b. Many important data organization functions such as sorting algorithms and duplicate detection or elimination algorithms rely on linear orders. It is not necessary to introduce linear order at every type because linear order at base types can be lifted, using a technique introduced to me by Libkin in our paper [130]. This section studies the effect of linear orders on conservative extension properties.

Lifting of linear orders

Recall that the Hoare ordering $\sqsubseteq^\flat$ on the subsets of an ordered set is defined as $X \sqsubseteq^\flat Y$ if and only if for every $x \in X$ there is $y \in Y$ such that $x \sqsubseteq y$. Then

Proposition 3.3.1 *Let $(D, \sqsubseteq)$ be a partially ordered set. Define an order $\lesssim^\flat$ on the finite subsets of D as follows: $X \lesssim^\flat Y$ if and only if either $X \sqsubseteq^\flat Y$ and $Y \not\sqsubseteq^\flat X$, or $X \sqsubseteq^\flat Y$ and $Y \sqsubseteq^\flat X$ and $X - Y \sqsubseteq^\flat Y - X$. Then $\lesssim^\flat$ is a partial order. Moreover, if $\sqsubseteq$ is a linear order, then so is $\lesssim^\flat$.*

Proof. The proof is by Libkin and can be found in [130]. □

Kupert, Saake, and Wegner [125] gave three linear orderings on collection types in their study of duplicate detection and elimination. The ordering defined above coincides with one of them. Incidentally, the above formulation is a special case of an order frequently used in universal algebra and combinatorics (see Kruskal [121] or Wechler [200]). An important feature of this technique of lifting linear orders is that the resulting linear orders are readily

seen to be computable by my very limited language.

Theorem 3.3.2 *When augmented with linear orders at all base types,* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ *can express linear orders* $\leq^s : s \times s \to \mathbb{B}$ *at all types* s.

Proof. Proceed by induction on s.

- $\leq^b$ is the given linear order on base type b.
- $x \leq^{s \times t} y \triangleq \mathit{if}\ \pi_1\ x \leq^s \pi_1\ y\ \mathit{then}\ (\mathit{if}\ \pi_1\ x =^s \pi_1\ y\ \mathit{then}\ \pi_2\ x \leq^t \pi_2\ y\ \mathit{else}\ \mathit{true})\ \mathit{else}\ \mathit{false}$
- $X \leq^{\{s\}} Y \triangleq \mathit{if}\ X \sqsubseteq^\flat_s Y\ \mathit{then}\ (\mathit{if}\ Y \sqsubseteq^\flat_s X\ \mathit{then}\ X \lesssim^\flat_s Y\ \mathit{else}\ \mathit{true})\ \mathit{else}\ \mathit{false}$, where
- $X \sqsubseteq^\flat_s Y \triangleq (\sum\{\!|(\mathit{if}\ (\sum\{\!|(\mathit{if}\ x \leq^s y\ \mathit{then}\ 1\ \mathit{else}\ 0) \mid y \in Y|\!\})) = 0\ \mathit{then}\ 1\ \mathit{else}\ 0) \mid x \in X|\!\}) = 0$ and
- $X \lesssim^\flat_s Y \triangleq (\sum\{\!|\mathit{if}\ x \in^s Y\ \mathit{then}\ 0\ \mathit{else}\ (\mathit{if}\ (\sum\{\!|\mathit{if}\ y \in^s X\ \mathit{then}\ 0\ \mathit{else}\ (\mathit{if}\ x \leq^s y\ \mathit{then}\ 1\ \mathit{else}\ 0) \mid y \in Y|\!\}) = 0\ \mathit{then}\ 1\ \mathit{else}\ 0) \mid x \in X|\!\}) = 0$. □

Hence the language endowed with linear orders at base types is denoted $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$.

Power of linear orders

Several queries commonly encountered in practical database environment but cannot be expressed in first-order logic can now be expressed. For example, find those rows in R whose first column value is maximum is definable as $\mathit{maxrows}(R) \triangleq \bigcup\{\mathit{if}\ (\sum\{\!|\ \mathit{if}\ \pi_1(x) = \pi_1(y)\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ \pi_1(y) \leq \pi_1(x)\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid x \in R|\!\} = 0)\ \mathit{then}\ \{y\}\ \mathit{else}\ \{\} \mid y \in R\}$. Another example is to find the rows in R whose first column value occurs most frequently by $\mathit{moderows}(R) \triangleq \mathit{maxrows}(\bigcup\{\{(\sum\{\!|\ \mathit{if}\ \pi_1(y) = \pi_1(x)\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid y \in R|\!\},\ x)\} \mid x \in R\})$. The language also has sufficient power to test whether the cardinality of a set R is odd or even by defining $\mathit{odd}(R) \triangleq \bigcup\{\mathit{if}\ \sum\{\!|\mathit{if}\ x \leq y\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid y \in R|\!\} = \sum\{\!|\mathit{if}\ y \leq x\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid y \in R|\!\}\ \mathit{then}\{()\}\ \mathit{else}\ \{\} \mid x \in R\} = \{()\}$.

More significantly, it can compute the rank assignment function. The definability of rank assignment leads to very unexpected conservativeness results to be discussed shortly.

Proposition 3.3.3 *A rank assignment function* $sort^s : \{s\} \to \{s \times \mathbb{Q}\}$ *is the function such that* $sort\{o_1, \ldots, o_n\} = \{(o_1, 1), \ldots, (o_n, n)\}$ *where* $o_1 < \ldots < o_n$. $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ *can define* $sort^s$.

Proof. The rank assignment function can be defined as $sort(R) \triangleq \bigcup\{\{(x, \sum\{\!|\, \mathit{if}\ y \leq x\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid y \in R \,|\!\}) \} \mid x \in R\}$. □

Linear orders lead to uniformity

The ability to compute a linear order at all types can be used to provide a more uniform proof of the conservative extension theorem. To illustrate this, let me introduce three partially interpreted primitives ι, $\odot$ and $\prod$ to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$, where b is some fixed type, $\odot : b \times b \to b$ is a commutative and associative binary operation, $\iota : b$ is the identity for $\odot$, and $\prod\{\!|e \mid x^s \in \{o_1, \ldots, o_n\}|\!\} = e[o_1/x^s] \odot \ldots \odot e[o_n/x^s] \odot \iota$ for any set $\{o_1, \ldots, o_n\}$ of type $\{s\}$. As an example, take $\odot$ to be $\cdot$ and b to be $\mathbb{Q}$, then ι becomes 1 and $\prod$ becomes the bounded product.

Theorem 3.3.4 *For every* i, o, *and* $h \geq \max(i, o, ht(b))$, $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \odot, \prod, \iota)_{i,o,h}$ *coincides with* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \odot, \prod, \iota)_{i,o,h+1}$.

Proof. It suffices to append the rules below to the rewrite system of the previous section. Note the use of the linear ordering $\leq$. The earlier rules on $\sum\{\!|e_1 \mid x \in e_2|\!\}$ can be replaced using these rules too, achieving conservative extension without needing $\div$. (If $\odot$ is also idempotent, then rules mirroring those for $\bigcup\{e_1 \mid x \in e_2\}$ can be used.)

- $\prod\{\!|e \mid x \in \{\}|\!\} \rightsquigarrow \iota$
- $\prod\{\!|e \mid x \in \{e'\}|\!\} \rightsquigarrow e[e'/x]$

- $\prod\{\!|e \mid x \in e_1 \cup e_2|\!\} \rightsquigarrow \prod\{\!|e \mid x \in e_1|\!\} \odot \prod\{\!|\mathit{if}\ x \in e_1\ \mathit{then}\ \iota\ \mathit{else}\ e \mid x \in e_2|\!\}$
- $\prod\{\!|e \mid x \in \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3|\!\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \prod\{\!|e \mid x \in e_2|\!\}\ \mathit{else}\ \prod\{\!|e \mid x \in e_3|\!\}$
- $\prod\{\!|e \mid x \in \bigcup\{\!|e_1 \mid y \in e_2|\!\}|\!\} \rightsquigarrow \prod\{\!|\prod\{\!|\mathit{if}\ (\sum\{\!|\mathit{if}\ x \in e_1[w/y]\ \mathit{then}\ (\mathit{if}\ w = y\ \mathit{then}\ 0\ \mathit{else}\ (\mathit{if}\ w \leq y\ \mathit{then}\ 1\ \mathit{else}\ 0))\ \mathit{else}\ 0 \mid w \in e_2|\!\}) = 0\ \mathit{then}\ e\ \mathit{else}\ \iota \mid x \in e_1|\!\} \mid y \in e_2|\!\}$ □

Linear orders lead to surprises

The two preceeding results have some surprising consequences. Let me proceed by adding for every complex object type s, the following primitives: $tc^s : \{s \times s\} \to \{s \times s\}$; $bfix^s(f,g) : \{s\}$, where $g : \{s\}$ and $f : \{s\} \to \{s\}$; and $powerset^s : \{s\} \to \{\{s\}\}$. The interpretation is that $tc(R)$ computes the transitive closure of R; $bfix(f,g)$ computes the bounded fixpoint of f with respect to g (that is, it is the least fixpoint of the equation $f(R) = g \cap (R \cup f(R))$); and $powerset(R)$ is the powerset of R.

Corollary 3.3.5 *The following languages have the conservative extension property:*

- $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, tc)$ *with displacement* 0 *and fixed constant* 1*;*
- $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, bfix)$ *with displacement* 0 *and fixed constant* 1*; and*
- $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, powerset)$ *with displacement* 0 *and fixed constant* 2*.*

Proof. The proof of the first one is given below, the other two are straightforward adaptation of the same technique. First observe that $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, tc^{\mathbb{Q}})$, where only the primitive for transitive closure on rational numbers is added, has the conservative extension property with displacement 0 and constant 1. Therefore, it suffices to show that tc^s is expressible in it for every s. We do so by exploiting the *sort* function by defining

- $tc(R) \triangleq decode(tc^{\mathbb{Q}}(encode(R, sort(dom(R)))), sort(dom(R)))$, where

- $dom(R) \triangleq \bigcup\{\{\pi_1\ x\} \mid x \in R\} \cup \bigcup\{\{\pi_2\ x\} \mid x \in R\}$,
- $encode(R,C) \triangleq \bigcup\{\bigcup\{\bigcup\{$*if* $\pi_1\ x = \pi_1\ y$ *then if* $\pi_2\ x = \pi_1\ z$ *then* $\{(\pi_2\ y,\ \pi_2\ z)\}$ *else* $\{\}$ *else* $\{\} \mid z \in C\} \mid y \in C\} \mid x \in R\}$, and
- $decode(R,C) \triangleq \bigcup\{\bigcup\{\bigcup\{$*if* $\pi_1\ x = \pi_2\ y$ *then if* $\pi_2\ x = \pi_2\ z$ *then* $\{(\pi_1\ y,\ \pi_1\ z)\}$ *else* $\{\}$ *else* $\{\} \mid z \in C\} \mid y \in C\} \mid x \in R\}$.

The purpose of $encode(R,C)$ is to produce a relation R' of rational numbers by replacing every pair $(o_1, o_2) \in R$ with a pair (n_1, n_2), where n_i is the rank of o_i in the rank table C. The purpose of $decode(R',C)$ is to recover, from the pair of ranks $(n_1, n_2) \in R'$, the pair (o_1, o_2) by looking up the rank table C. Therefore, $tr(R)$ is computed by first encoding R into a binary relation R' of rational numbers, then compute $tr^{\mathbb{Q}}(R')$, and finally recovering from it the transitive closure of R. □

Conservativity of $\mathcal{NRC}(=, powerset)$ was considered by Hull and Su [99] and Grumbach and Vianu [79]. The former showed that $\mathcal{NRC}(=, powerset)_{i,o,h} \neq \mathcal{NRC}(=, powerset)_{i,o,h+1}$ for any h and $i = o = 1$, implying the failure of conservative extension for $\mathcal{NRC}(=, powerset)$ with respect to flat relations. The latter generalized this result to relations of any height. Corollary 3.3.5 above shows that the failure at height higher than 2 can be repaired by augmenting $\mathcal{NRC}(=, powerset)$ with a summation operator, some limited arithmetic operations, and linear orders at base types.

More recently, Suciu [178] showed, using a technique related to that of Van den Bussche [56], that $\mathcal{NRC}(=, bfix)_{i,o,h} = \mathcal{NRC}(=, bfix)_{i,o,h+1}$ for $i = o = 1$. This result is remarkable because he did not need any arithmetic operation. Corollary 3.3.5 above shows that the conservativity of bounded fixpoint can be extended to all input and output in the presence of summation.

Immerman [103] showed that first-order logic with least fixpoint operator (*lfp*) and order computes exactly the class of queries that have polynomial time complexity. This result may imply $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, lfp)_{1,1,h} = \mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, lfp)_{1,1,h+1}$. In

that case, $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, lfp)$ is conservative over flat relations. This result should be contrasted with Corollary 3.3.5 above. The languages there do not necessarily give us all polynomial time queries over flat relations. Furthermore, conservativity holds for them over any input and output. As evidence that the languages do not necessarily compute all polynomial time queries, I observe that every predicate $p : \mathbb{Q} \to \mathbb{B}$ expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ is either finite or cofinite; see Section 4.2.

3.4 Internal generic functions preserve conservative extension properties

Internal generic functions

There is a more general conservative extension result underlying Corollary 3.3.5. To describe precisely this result, I introduce type variables α_i and consider nonground complex object types

$$\sigma, \tau ::= \alpha \mid b \mid unit \mid \sigma \times \tau \mid \{\sigma\}$$

If α_1, ..., α_n occur in σ, then $\sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$ stands for the type obtained by replacing every occurrence of α_i in σ by s_i. A complex object type s is an instance of a nonground complex object type σ if there are complex object types s_1, ..., s_n such that $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$ where α_1, ..., α_n are all the type variables in σ. The minimal height $mht(\sigma)$ of type σ is defined as the depth of nesting of set brackets in σ. That is, $mht(\sigma)$ is equivalent to $ht(s)$ where s is obtained from σ by replacing all occurrences of type variables in σ by some base types b. I write $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ for the family of functions $p^{s_1,\ldots,s_n} : s \to t$ where $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$ and $t = \tau[s_1/\alpha_1, \ldots, s_n/\alpha_n]$. (Note that for each s_1, ..., s_n, there is exactly one $p^{s_1,\ldots,s_n}$ in the family $p^{\alpha_1,\ldots,\alpha_n}$.) The minimal height $mht(p)$ of $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ is defined as $\max(mht(\sigma), mht(\tau))$.

Let $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n, t/\alpha]$. Let $dom^{s,t}_{\sigma,\alpha}(o)$ be the set of subobjects of type t in the object $o : s$ occurring at positions corresponding to the type variable α. Formally,

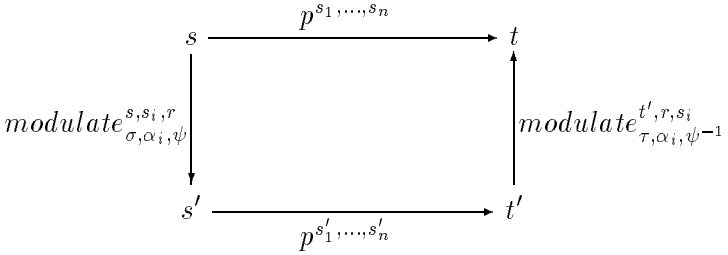
define $dom^{s,t}_{\sigma,\alpha} : s \to \{t\}$ as follows: $dom^{s,t}_{b,\alpha}(x) = \{\}$; $dom^{s,t}_{\alpha,\alpha}(x) = \{x\}$; $dom^{s,t}_{\alpha',\alpha}(x) = \{\}$, where α and α' are distinct type variables; $dom^{u \times v,t}_{\sigma \times \tau,\alpha}(x,y) = dom^{u,t}_{\sigma,\alpha}(x) \cup dom^{v,t}_{\tau,\alpha}(y)$; and $dom^{\{s\},t}_{\{\sigma\},\alpha}(X) = \bigcup\{dom^{s,t}_{\sigma,\alpha}(x) \mid x \in X\}$.

Definition 3.4.1 The family of functions $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ is **internal** (see Hull [98]) in α_i if for all complex object types $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$, $t = \tau[s_1/\alpha_1, \ldots, s_n/\alpha_n]$, and complex object $o : s$, it is the case that $dom^{t,s_i}_{\tau,\alpha_i}(p^{s_1,\ldots,s_n}(o)) \subseteq dom^{s,s_i}_{\sigma,\alpha_i}(o)$. □

In other words, $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ is internal in α_i if it does not invent new values in positions corresponding to the type variable α_i. That is, every subobject in $p(O)$ at a position corresponding to α_i can also be found in O at a position corresponding to α_i.

Let $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n, t/\alpha]$. $r = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n, t'/\alpha]$, and $\psi : t \to t'$. Let $modulate^{s,t,t'}_{\sigma,\alpha,\psi}(O)$ be the object $O' : r$ obtained by replacing every subobject $o : t$ in $O : s$ occurring in positions corresponding to type variable α by $\psi(o) : t'$. Formally, define $modulate^{s,t,t'}_{\sigma,\alpha,\psi} : s \to r$ as follows: $modulate^{s,t,t'}_{b,\alpha,\psi}(x) = x$; $modulate^{s,t,t'}_{\alpha,\alpha,\psi}(x) = \psi(x)$; $modulate^{s,t,t'}_{\alpha',\alpha,\psi}(x) = x$, where α and α' are distinct type variables; $modulate^{u \times v,t,t'}_{\sigma \times \tau,\alpha,\psi}(x,y) = (modulate^{u,t,t'}_{\sigma,\alpha,\psi}(x), modulate^{v,t,t'}_{\tau,\alpha,\psi}(y))$; $modulate^{\{s\},t,t'}_{\{\sigma\},\alpha,\psi}(X) = \{modulate^{s,t,t'}_{\sigma,\alpha,\psi}(x) \mid x \in X\}$.

Definition 3.4.2 The family of functions $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ is **generic** in α_i if for all complex object types $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$, $t = \tau[s_1/\alpha_1, \ldots, s_n/\alpha_n]$, complex object $o : s$, set $R : \{r\}$, and $\psi : s_i \to r$ such that ψ is a bijection from $dom^{s,s_i}_{\sigma,\alpha_i}(o)$ to R and $\psi^{-1} : r \to s_i$ is its inverse when restricted to $dom^{s,s_i}_{\sigma,\alpha_i}(o)$, it is the case that



the diagram above, where $s'_j = s_j$ for $j \neq i$ and $s'_i = r$, commutes. □

The aim of this section is to show that adding a family $p^{\alpha_1,\ldots,\alpha_n}$, internal and generic in

all type variables, to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ does not destroy its conservative extension property.

An implication of internal genericity

This is best seen if the linear orders assumed for base types are well-founded. I assume for now that $\leq^b : b \times b \to \mathbb{B}$ is a well-founded linear order for every base type b. Note that, for the rest of this section, I use $\leq^{\mathbb{Q}}$ to stand for this well-founded linear order on rational numbers and use $min^{\mathbb{Q}}$ to denote the rational number that is least with respect to this well-found linear order. Consider $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup)$ obtained by adding the construct depicted in Figure 3.2 to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$.

$$\frac{e_1 : s \quad e_2 : \{t\}}{\bigsqcup\{e_1 \mid x^t \in e_2\} : s}$$

Figure 3.2: The $\bigsqcup$-construct.

The expression $\bigsqcup\{e_1 \mid x^t \in e_2\}$ denotes the greatest element in the set $\{e_1 \mid x^t \in e_2\}$ (it is min^s when the set is empty). I write min^s as a shorthand for the least element of type s with respect to $\leq^s$; hence, $min^{s \times t}$ is (min^s, min^t) and $min^{\{s\}}$ is $\{\}$. Note that $\bigsqcup\{e_1 \mid x \in e_2\}$, where $e_1 : \{s\}$, is already definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ and can be treated as a syntactic sugar. It is clear that both $dom^{s,t}_{\sigma,\alpha}$ and $modulate^{s,t,t'}_{\sigma,\alpha,\psi}$ are definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup)$ whenever ψ is.

Proposition 3.4.3 *Let $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ be a family of functions that is internal generic. Then $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup)$ endowed with the family of primitives $p^{\alpha_1,\ldots,\alpha_n}$ has precisely the expressive power of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup)$ endowed with just the primitive $p^{\mathbb{Q},\ldots,\mathbb{Q}}$.*

Proof. For each $s = \sigma[s_1/\alpha_1, \ldots, s_n/\alpha_n]$ and $o : \{s_i\}$, define

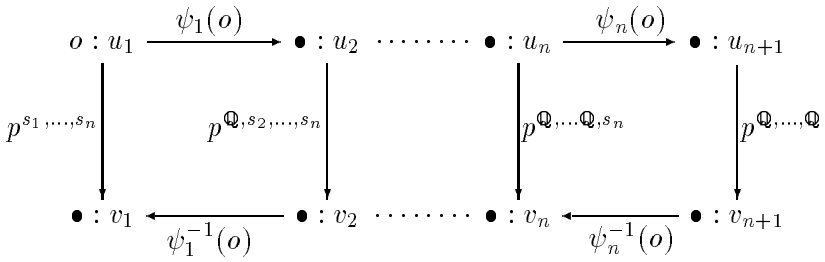
- $\psi(o) \triangleq \lambda x. \bigsqcup\{\mathit{if}\ x = \pi_1\ y\ \mathit{then}\ \pi_2\ y\ \mathit{else}\ 0 \mid y \in \mathit{sort}(o)\}$ and
- $\psi^{-1}(o) \triangleq \lambda x. \bigsqcup\{\mathit{if}\ x = \pi_2\ y\ \mathit{then}\ \pi_1\ y\ \mathit{else}\ \mathit{min}^s \mid y \in \mathit{sort}(o)\}$,

where $\mathit{sort} : \{s_i\} \to \{s_i \times \mathbb{Q}\}$ is as defined in Corollary 3.3.3. $\psi(o)$ and $\psi^{-1}(o)$ are functions of type $s_i \to \mathbb{Q}$ and $\mathbb{Q} \to s_i$ respectively. Clearly, $\psi(o)$ when restricted to o is a bijection whose inverse is $\psi^{-1}(o)$.

Let $u_i = \sigma[\mathbb{Q}/\alpha_1, \ldots, \mathbb{Q}/\alpha_{i-1}, s_i/\alpha_i, \ldots, s_n/\alpha_n]$ and $v_i = \tau[\mathbb{Q}/\alpha_1, \ldots, \mathbb{Q}/\alpha_{i-1}, s_i/\alpha_i, \ldots, s_n/\alpha_n]$. Note that $s = u_1$ and $t = v_1$. Define

- $\psi_i(o) \triangleq \mathit{modulate}^{u_i, s_i, \mathbb{Q}}_{\sigma, \alpha_i, \psi(\mathit{dom}^{s, s_i}_{\sigma, \alpha_i}(o))}$ and
- $\psi_i^{-1}(o) \triangleq \mathit{modulate}^{v_{i+1}, \mathbb{Q}, s_i}_{\tau, \alpha_i, \psi^{-1}(\mathit{dom}^{s, s_i}_{\sigma, \alpha_i}(o))}$.

Then the following diagram commutes by induction on n and by the assumption that the family $p^{\alpha_1, \ldots, \alpha_n}$ is internal and generic.



Hence $p^{s_1, \ldots, s_n} = \lambda x. \psi_1^{-1}(x) \circ \cdots \circ \psi_n^{-1}(x) \circ p^{\mathbb{Q}, \ldots, \mathbb{Q}} \circ \psi_n(x) \circ \cdots \circ \psi_1(x)$. The right hand side is clearly expressible in $\mathcal{NRC}(\mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup, p^{\mathbb{Q}, \ldots, \mathbb{Q}})$. □

I now proceed to prove

THE CONSERVATIVENESS OF $\mathcal{NRC}(\mathbb{B},\ \mathbb{Q},\ +,\ \cdot,\ -,\ \div,\ \sum,\ =,\ \leq,\ \bigsqcup)$

Proposition 3.4.4 *$\mathcal{NRC}(\mathbb{B},\ \mathbb{Q},\ +,\ \cdot,\ -,\ \div,\ \sum,\ =,\ \leq,\ \bigsqcup)$ has the conservative extension*

property with fixed constant 0. *Moreover, when endowed with any additional primitive* p, *it retains the conservative extension property with fixed constant* $ht(p)$.

Proof. Add the following rewrite rules for $\bigsqcup$, assuming that the use of the construct $\bigsqcup\{e_1 \mid x \in e_2\}$ is restricted to the situation when the type of e_1 is not a set type (when $e_1 : \{s\}$, it is treated as a shorthand.)

- $\bigsqcup\{e \mid x \in \{\}\} \rightsquigarrow min$
- $\bigsqcup\{e_1 \mid x \in \{e_2\}\} \rightsquigarrow e_1[e_2/x]$
- $\bigsqcup\{e \mid x \in e_1 \cup e_2\} \rightsquigarrow$ *if* $\bigsqcup\{e \mid x \in e_1\} \leq \bigsqcup\{e \mid x \in e_2\}$ *then* $\bigsqcup\{e \mid x \in e_2\}$ *else* $\bigsqcup\{e \mid x \in e_1\}$
- $\bigsqcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \rightsquigarrow \bigsqcup\{\bigsqcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$
- $\bigsqcup\{e \mid x \in$ *if* e_1 *then* e_2 *else* $e_3\} \rightsquigarrow$ *if* e_1 *then* $\bigsqcup\{e \mid x \in e_2\}$ *else* $\bigsqcup\{e \mid x \in e_3\}$
- $\pi_i \bigsqcup\{e_1 \mid x \in e_2\} \rightsquigarrow \bigcup\{$*if* $\sum\{$*if* $e_1 \leq e_1[y/x]$ *then* 1 *else* 0 $\mid y \in e_2\} =$ 1 *then* $\pi_i\ e_1$ *else* $\{\} \mid x \in e_2\}$, when $e_1 : \{s\}$.
- $\pi_i \bigsqcup\{e_1 \mid x \in e_2\} \rightsquigarrow \bigsqcup\{$*if* $\sum\{$*if* $e_1 \leq e_1[y/x]$ *then* 1 *else* 0 $\mid y \in e_2\} =$ 1 *then* $\pi_i\ e_1$ *else* $\{\} \mid x \in e_2\}$, when e_1 is not of set type.

The extended collection of rewrite rules forms a weakly normalizing rewrite system and conservativity can be derived by induction on the induced normal forms along the lines of Theorem 3.1.4. □

Putting together the two previous propositions, the desired theorem follows straightforwardly.

Theorem 3.4.5 $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq, \bigsqcup)$ *endowed with an internal generic family* $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$ *has the conservative extension property with fixed constant* $mht(p)$. □

As remarked earlier, $\bigsqcup\{e_1 \mid x \in e_2\}$ is already definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ if $e_1 : \{s\}$. Therefore, if every type variable occurs in the scope of some set brackets in σ and τ, then the assumption of well-foundedness on $\leq^b$ used in Proposition 3.4.3 is not required and the proposition holds for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$. Thus, we have

Corollary 3.4.6 *$\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ endowed with an internal generic family $p^{\alpha_1,\ldots,\alpha_n} : \sigma \to \tau$, where each type variable is within the scope of some set brackets, has the conservative extension property at all input and output heights with fixed constant $mht(p)$.* □

In particular, any polymorphic function definable in the algebra of Abiteboul and Beeri [2], which is equivalent to $\mathcal{NRC}(=, powerset)$, gives rise to an internal generic family of functions for all possible instantiations of type variables. Since the Abiteboul and Beeri algebra has the power of a fixpoint logic, a great deal of polymorphic functions can be added to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ without destroying its conservative extension property (but may be increasing the fixed constant). Corollary 3.3.5 is special case of this general result.

Chapter 4

Finite-cofinite Properties

All popular commercial database query languages such as SQL are equipped with aggregate functions and linear orders on numbers. These languages are further complicated by the fact that they may use bag semantics as well as set semantics. Theoretical results obtained on the basis of first-order logic or flat relational algebra, as in Chandra and Harel [38] and Fagin [61], often do not apply to these real query languages. For example, while it is known [7] that transitive closure is inexpressible in first-order logic, its inexpressibility in SQL is not clear. Indeed, it is not even clear how one can stretch first-order logic so as to embed aggregate functions in it naturally.

Recently there is an increasing interest to study query languages which more closely approximates real query languages. Chaudhuri and Vardi [40]; Albert [8]; Grumbach and Milo [77] and Grumbach, Milo, and Kornatzky [78] all consider query languages for bags. Mumick, Pirahesh, and Ramakrishnan [147], and Libkin and myself [132] all consider query languages with aggregate functions. Libkin and I [135] provided an explicit connection between bags and aggregate functions, via which many results proved for aggregate functions can be transferred to bags and vice versa.

Grumbach and Milo put forward two questions on bag query languages in their paper [77]. The first is whether their bag query language can express the parity test on the cardinality

of sets without using any power operators. The second is whether their bag query language can express transitive closure on relations without using any power operators. Paredaens posed to me a third question on bag query languages in a conversation at Bellcore [158]. His question was whether the test for balanced binary trees is expressible in Libkin and my bag query language.

$\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ is an arguably natural extension of $\mathcal{NRC}(\mathbb{B}, =)$ with aggregate functions. Moreover, it possesses the conservative extension property which has a simplifying effect on the analysis of the expressive power of the language. In this chapter, I demonstrate this simplifying effect by proving several finite-cofiniteness properties for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ by analysing the normal forms induced by the conservative extension properties. The last of these properties yields negative solutions of all the above conjectures as immediate corollaries.

Organization

Section 4.1. Every property expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ on rational numbers is shown either to hold for finitely many rational numbers or to fail for finitely many rational numbers. This result is a generalization of the classic result that, in the language of pure identity, first-order logic can only express properties that are finite or cofinite. A corollary of this result is that, inspite of its arithmetic power, $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ cannot test whether one number is bigger than another number. This result justifies the augmentation of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ with linear orders on base types.

Section 4.2. Every property expressible in the augmented language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ on natural numbers is again shown to be finite or cofinite. Many consequences follow from this result, including the inexpressibility of parity test in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ on natural numbers. This result is a very strong evidence that the conservative extension theorem for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ is not a consequence of Immerman's result on fixpoint queries in the presence of linear orders.

Section 4.3. Certain classes of graphs are introduced. Expressibility of properties on these graphs in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ when the linear order is restricted to rational numbers is considered. I show that these properties are again finite-cofinite. This result settles the conjectures of Grumbach and Milo [77] and Paredaens [158] that parity-of-cardinality test, transitive closure, and balanced-binary-tree test cannot be expressed with aggregate functions or with bags. This also generalizes the classic result of Aho and Ullman [7] that flat relational algebra cannot express transitive closure to a language which is closer in strength to SQL.

4.1 Finite-cofiniteness of predicates on rational numbers

It is well known that in the pure language of identity (that is, with no predicate symbols other than equality), first-order logic can only express properties that are finite or cofinite. This fact can be extended to fixpoint logic via pebble games [55]. As an example of the theoretical usefulness of the conservative extension theorem on $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$, I show below that $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ can only express properties on rational numbers that are finite or cofinite.

Proposition 4.1.1 *Let $p : \mathbb{Q} \to \mathbb{B}$ be a primitive predicate on rational numbers. Suppose p is definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$. Then p must be finite or cofinite. That is, either there are only finitely many rational numbers which satisfy p or there are only finitely many rational numbers which do not satisfy p.*

Proof. Let p be definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$. By Theorem 3.2.4, it must be defined by a normal form $\lambda x.e$ of height $ht(\mathbb{Q} \to \mathbb{B}) = 0$. Thus e must be constructed entirely from constants, $+$, $-$, $\div$, $\cdot$, $=^b$, and *if-then-else*.

First add $\wedge$, $\vee$, and $\neg$ (with the usual interpretation) to the language. Rewrite e into a formula without *if-then-else* such that all the leaves are of the form $A = B$, where A and B uses just rational constants, $+$, $-$, $\cdot$, and $\div$. This step can be accomplished using rules

such as:

- $\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \leadsto (e_1\ \wedge\ e_2)\ \vee\ ((\neg\ e_1)\ \wedge\ e_3)$, where $e_2 : \mathbb{B}$ and $e_3 : \mathbb{B}$.
- $(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \cdot e_4 \leadsto \mathit{if}\ e_1\ \mathit{then}\ e_2 \cdot e_4\ \mathit{else}\ e_3 \cdot e_4$

Each leaf $A = B$ of the outcome of the previous step is turned into a polynomial equation $C = 0$ where C may use only $\cdot$, $+$, $-$, and constants, but not $\div$. This is achieved using rules like:

- $A \div B = C \leadsto A = C \cdot B$
- $A \cdot (B + C) = D \leadsto (A \cdot B) + (A \cdot C) = D$

The proposition follows immediately from the claim below.

Claim. Let $E : \mathbb{B}$ be any formula, of one free variable $x : \mathbb{Q}$, constructed entirely from x, $\wedge$, $\vee$, $\neg$, rational constants, $+$, $-$, and $\cdot$ such that the leaves of E are polynomial equations of the form $C = 0$. Then either $E[n/x]$ is true of finitely many rationals n or it is false of finitely many rationals n.

Proof of Claim. Proceed by structural induction on E.

- Suppose E is $C = 0$. It is well known that polynomials of degree k has at most k roots. Hence there are only finitely many n for which $C[n/x] = 0$ is true.
- Suppose E is $\neg\ E'$. By hypothesis, either $E'[n/x]$ is true for finite many n or it is false for finitely many n. In the first case, $E[n/x]$ is false for finitely many n. In the second case, $E[n/x]$ is true for finitely many n.
- Suppose E is $E_1 \wedge E_2$. By hypothesis, either there are finitely many n so that $E_1[n/x]$ is true or there are finitely many n so that $E_1[n/x]$ is false. In this first case, it is clear that there are only finitely n so that $E[n/x]$ is true. For the second case, there are two subcases. The first subcase, suppose the hypothesis on E_2 yields $E_2[m/x]$ is true

only for finitely many m. This implies $E[m/x]$ holds only for finitely many m. The other subcase is when the hypothesis on E_2 yields $E_2[m/x]$ is false only for finitely many m. So $E[n/x]$ is false only for finitely many n.

- Suppose E is $E_1 \vee E_2$. This case follows because $E_1 \vee E_2$ if and only if $\neg((\neg E_1) \wedge (\neg E_2))$. □

Given any rational number, there are both infinitely many rational numbers greater than it and infinitely many rational numbers less than it. Therefore, the usual linear order $\leq^{\mathbb{Q}}: \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ on rational numbers cannot be expressed in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$, in spite of its arithmetic prowess.

Corollary 4.1.2 *$\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ cannot define $\leq^{\mathbb{Q}}$.* □

4.2 Finite-cofiniteness of predicates on natural numbers

Corollary 4.1.2 justifies augmenting $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ with linear orders. The augmented language is indeed a very much richer language. As shown in Section 3.3, $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ can even test whether the cardinality of a set is odd or even. This fact is significant because this query cannot be expressed in first-order logic. This language still has the finite-cofiniteness property, when restricted to natural numbers.

Proposition 4.2.1 *Let $p : \mathbb{Q} \to \mathbb{B}$ be any predicate on rational numbers. Suppose p is expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$. Then either p holds for finitely many natural numbers or p fails for finitely many natural numbers. That is, the restriction of p to $\mathbb{N}$ is either finite or is cofinite.*

Proof. The trick is to realize that p has height 0. Thus it is definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ using an expression of height 0; see Theorem 3.4.6. It is then straightforward to modify the proof of Proposition 4.1.1 to obtain a proof for this proposition. We need

only to deal with the new case of E being $C \leq 0$. Since every polynomial equation of degree k has at most k roots, let n be the largest root for C. Then either for all $m > n$, $C[m/x] < 0$; that is, $C[n/x] \leq 0$ fails for finitely many n. Or for all $m > n$, $C[m/x] > 0$; that is, $C[n/x] \leq 0$ holds for finitely many n. An earlier proof by Libkin based on the same trick can be found in our paper [130]. □

As a result, while $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ can test whether the cardinality of a set is odd or even, it cannot test whether a rational number is actually an odd natural number or not.

Corollary 4.2.2 *Let $p : \mathbb{Q} \to \mathbb{B}$ be a predicate such that $p(n)$ holds if and only if n is a odd natural number. Then p is not expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$.* □

Libkin and I [130] introduced a query language for bags by interpreting the syntax of $\mathcal{NRC}$ bag-theoretically. This language is equivalent to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ minus the division operator. This relationship, proved by Libkin and myself [130], gives rise to some interesting corollaries. The first is a consequence of Corollary 4.2.2: the basic query language for bags introduced by Libkin and myself [130] can test whether a bag contains an odd number of distinct objects, but it cannot test whether a bag contains an odd number of objects. A special case of this result was independently proved by Grumbach and Milo [77]. The second is a consequence of Proposition 4.2.1: the basic bag language of [130] can only express those predicates $p : \{|unit|\} \to \mathbb{B}$, where $\{|unit|\}$ is the type of objects that are bags of *unit*, that are either finite or cofinite. This result is a generalization of the first consequence (and hence of Grumbach and Milo's result). The third is a consequence of Theorem 3.4.6: the basic language for bags introduced by Libkin and myself [130] has the conservative extension property at all input and output heights with constant 0; however, the displacement is 1 due to the translations used. See my paper with Libkin [130] for details. The bag language of Grumbach and Milo minus its power operators is equivalent to Libkin and mine. Hence the above discussion applies to this fragment of their language too.

4.3 Finite-cofiniteness of predicates on special graphs

The two finite-cofiniteness theorems presented earlier are straightforward consequences of two observations. The first observation is that the predicates involved have height 0. My conservative extension theorems immediately tell us that these predicates can be implemented using expressions of height 0 and hence no set is involved. The second observation is that such expressions are essentially boolean combinations of polynomial equations. The fundamental theorem of analysis tells us such equations have finite number of roots. Finite-cofiniteness then follows without complication.

Predicates of height 0 are simple from a database perspective because they concern primarily the base types. Predicates on graphs are seen more frequently in database query languages, first-order logics, and finite models. These predicates are of height 1 and hence they involve sets. They are considerably more difficult to analyse and hence they are very interesting.

In this section, the expressive power of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ over unordered graphs is considered. The language $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ is obtained by adding the usual linear order on rational numbers to $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$. In particular, I show that every predicate $p : \{b \times b\} \to \mathbb{B}$ definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$, when restricted to certain classes of unordered graphs, either holds for finitely many non-isomorphic graphs or fails for finitely many non-isomorphic graphs. As the technique applied on this problem is sophisticated, I first present the eureka step before I present the proof details. After that, I demonstrate the application of this result to the conjectures of Grumbach, Milo, and Paredaens.

An insight into the structure of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ queries

$\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ can construct arbitrarily deeply nested sets, and it can implement many aggregate functions. On the face of it, both of these features add complexity to the analysis of graph queries. It is fortunate that nested sets turn out to be a red herring because Theorem 3.2.4 holds for $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. That is,

$\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ has the conservative extension property. Since graph queries has height 1, it is only necessary for us to consider $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ expressions having height 1.

The rewriting done in the conservative extension theorem to eliminate intermediate data in fact gives us more than just expressions of height 1. It produces normal forms having a rather special trait. Let $e : \mathbb{Q}$ be an expression of height 1 in normal form. Let $R : \{b \times b\}$ be the only free variable in e. Let b be an unordered base type. Let e contains no constant of type b. Then e contains no subexpression of the form $\bigcup\{e_1 \mid x \in e_2\}$. Also, every subexpression involving $\sum$ is guaranteed to have the form $\sum\{\!|e_1 \mid x \in R|\!\}$.

It is natural to speculate on what e can look like. The most natural shape that comes to mind is the one depicted below.

$$\sum \left\{ \left| \ \ldots \sum \left\{ \left| \begin{array}{l} \mathit{if}\ P_1 \\ \mathit{then}\ f_1 \\ \vdots \\ \mathit{else\ if}\ P_h \\ \mathit{then}\ f_h \\ \mathit{else}\ f_{h+1} \end{array} \right| \ x_1 \in R \right\} \ldots \right| \ x_n \in R \right\}$$

Assume that the probability, in terms of the number of edges in R, of P_i being true and $P_{j<i}$ being false is p_i. Then the expression above is equivalent to the polynomial $N^n \cdot (p_1 \cdot f_1 + \ldots + p_{h+1} \cdot f_{h+1})$, with N being the number of edges in R.

This observation is a crucial for two reasons. First, the use of the summation operator is no longer arbitrary. It is now used only for computing the number of edges in R. All other uses of it have been replaced by a polynomial expression. Second, the expression no longer depends on the topology of the graph R. The only thing in R that can affect the value of the polynomial (and hence the original expression) is the cardinality of R. Then a result similar to Proposition 4.1.1 can be derived, leading to finite-cofiniteness of graph queries for which the probability assumption holds.

The insight above leads to a search for classes of graphs that possess sufficient regularity so that the required probability analysis can be performed. The simplest class of such graphs is probably the k-multi-cycles defined below.

Definition 4.3.1 *A binary relation* $O : \{b \times b\}$ *is called a* **k-multi-cycle** if it is nonempty and is of the form

$$\left\{ \begin{array}{ccc} (o_1^1, o_2^1), (o_2^1, o_3^1), & \ldots, & (o_{h-1}^1, o_h^1), (o_h^1, o_1^1), \\ \vdots & & \vdots \\ (o_1^m, o_2^m), (o_2^m, o_3^m), & \ldots, & (o_{h-1}^m, o_h^m), (o_h^m, o_1^m) \end{array} \right\}$$

where $h \geq k$ and o_i^j are all distinct. That is, it is a graph containing $m \geq 1$ unconnected cycles of equal length $h \geq k$. □

Let me first provide a sketch of how the probability anaylsis discussed earlier can be carried out on k-multi-cycles. Two preliminary definitons are needed for this purpose.

Define $distance_c(o, o', O)$ to be a predicate that holds if and only if the distance from node $\pi_1 o$ to node $\pi_2 o'$ in k-multi-cycle O is c. Note that $distance_c$ is definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =)$ for each constant c.

Define a *d-state* S with respect to variables $R : \{b \times b\}$, x_1, ..., $x_m : b \times b$ to be a conjunction of formulae of the form $distance_c(x_i, x_j, R)$ or the form $\neg distance_c(x_i, x_j, R)$ such that for each $0 \leq c \leq d$, $1 \leq i, j \leq m$, either $distance_c(x_i, x_j, R)$ or $\neg distance_c(x_i, x_j, R)$ must appear in it. Moreover, S has to be satisfiable in the sense that some chain O of length d and edges o_1, ..., o_m in O can be found so that $S[O/R, o_1/x_1, ..., o_m/x_m]$ holds.

Let $R : \{b \times b\}$, x_1, ..., $x_m : b \times b$ be fixed. Since any chain can be extended to a cycle, this implies that any d-state with respect to these variables can be satisfied by some d-multi-cycle. Conversely, if a k-multi-cycle is shorter than d, then it cannot satisfy every d-state with respect to these variables.

Proposition 4.3.2 *Let e be an expression of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ having $R : \{b \times b\}$, $N : \mathbb{Q}$, x_1, ..., $x_m : b \times b$ as free variables such that e has the special form below*

$$\sum \left\{ \left| \ \ldots \sum \left\{ \left| \begin{array}{l} \textit{if } P \\ \textit{then } E \\ \textit{else } 0 \end{array} \right| x_{m+1} \in R \right\} \ \ldots \ \right| x_{m+n} \in R \right\}$$

where E is a ratio of polynomials in terms of N, P is a boolean combination of formulae of the form $\pi_i x_{i'} = \pi_j x_{j'}$, $\pi_i x_{i'} \neq \pi_j x_{j'}$, $\neg distance_c(x_i, x_j, R)$, or $distance_c(x_i, x_j, R)$. Let $d \geq (n+m) \cdot (C+1)$ where C is the sum of the c's for each $distance_c(x_i, x_j, R)$ or $\neg distance_c(x_i, x_j, R)$ in P. Let S be any d-state with respect to R, x_1, ..., x_m. Then there is a ratio r of polynomials in terms of N such that for any d-multi-cycle O, and edges o_1, ..., o_m in O making $S[O/R, o_1/x_1, ..., o_m/x_m]$ true, it is the case that $e[O/R, o_1/x_1, ..., o_m/x_m, card(O)/N] = r[card(O)/N]$.

Proof. By the probability p for a predicate P of n free variables to hold with respect to a graph O, I mean the proportion of the instantiations of the free variables to edges in O that make P true. The key to the proof of this proposition is in realizing that the probability p for P to hold can be determined in the case of k-multi-cycle when k is large (any $k \geq d$ is good enough). Moreover p can be expressed as a ratio of two polynomials of N. Thus r can be defined as $N^n \cdot p \cdot E$

The probability p can be calculated as follows. First, generate all possible d-states D_j's with respect to the variables R, x_1, ..., x_{m+n}. Second, determine the probability q_j of D_j given the certainty of S; this can be calculated using the procedure to be given shortly. Third, eliminate those D_j's that are inconsistent with the conjunction of S and P. Finally, calculate p by summing the q_j's corresponding to those remaining d-states.

It remains to show that each q_i can be expressed as a ratio of two polynomials in N. Partition the positive leaves of the corresponding D_i into groups so that the variables in each group are connected between themselves and are unconnected with those in other groups. (Variables x and y are said to be connected in D_i if there is a positive leaf $distance_c(x, y, R)$ in D_i.) Note that the negative leaves merely assert that these groups are unconnected. Then we

proceed by induction on the number of groups.

The base case is when there is just one group. In such a situation, all the variables lie on the same cycle. Since a d-state can be satisfied by a chain of length d, these variables must lie on a line. Let u be the number of bound variables amongst x_{m+1}, ..., x_{m+n} appearing in the group; in this case $u = n$. Then $q_i = N \div N^u$ if no variables amongst x_1, ..., x_m appear in the group. Otherwise, $q_i = 1 \div N^u$. In either case, q_i is a ratio of polynomials in N.

For the induction case, suppose we have more than one group. The independent probability of each group can be calculated as in the base case. Then q_i is the difference between the product of these independent probabilities and the sum of the probabilities where these groups are made to overlap in all possible ways. These groups are made to overlap by turning some negative leaves in D_i into positive ones so that the results are again d-states. Notice that when groups overlap, the number of groups strictly decreases. Hence the induction hypothesis can be applied to obtain these probabilities as ratios of polynomials in N. Consequently, q_i can be expressed as a ratio of polynomials in N as desired. □

The proposition above shows that expressions of the given special form can be reduced to a simple polynomial in terms of the number of edges in R. In the theorem below, I sketch the process for converting any expression of type $\{b \times b\} \to \mathbb{B}$ in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ into this special form.

Theorem 4.3.3 *Let $G : \{b \times b\} \to \mathbb{B}$ be a function expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. Then there is some k such that for all k-multi-cycles O, it is the case that $G(O)$ is true; or for all k-multi-cycles O, it is the case that $G(O)$ is false.*

Proof. Let $G : \{b \times b\} \to \mathbb{B}$ be implemented by the $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ expression $\lambda R.E$. Without loss of generality, E can be assumed to be a normal form with respect to the rewrite system used in the proof of conservative extension theorem, Theorem 3.2.4. We note that such an E contains no subexpression of the form $\bigcup\{e_1 \mid x \in e_2\}$. Furthermore, all occurrences of summation in E must be of the form $\sum\{\!|e \mid x \in R|\!\}$.

Let us temporarily enrich our language with the usual logical operators $\vee$, $\wedge$, $\neg$, $\neq$, $\not\leq$, as well as $distance_c$ and $\neg distance_c$. Also introduce a new variable $N : \mathbb{Q}$, which is to be interpreted as the cardinality of R. Rewrite all summations into the special form given below

$$\sum \left\{ \left| \ \ldots \sum \left\{ \left| \begin{array}{l} if\ P \\ then\ f \\ else\ 0 \end{array} \right| x_{m+1} \in R \right\} \ \ldots \ \right| x_{m+n} \in R \right\}$$

so that f has the form $h \div g$, where h is a polynomial in terms of N and g is either a polynomial in terms of N or is again a subexpression of the same special form. Also, P is a formula whose leaves are of the following form: $\pi_i x_{i'} = \pi_j x_{j'}$, $\pi_i x_{i'} \neq \pi_j x_{j'}$, $distance_c(x_i, x_j, R)$, $\neg distance_c(x_i, x_j, R)$, $U =^{\mathbb{Q}} V$, $U \neq^{\mathbb{Q}} V$, $U \leq V$, or $U \not\leq V$, where U and V also have the same special form.

Let the resultant expression be F. The rewriting should be such that for all sufficiently long k-multi-cycles O, $F[O/R, card(O)/N]$ holds if and only if $E[O/R]$ holds. This rewriting can be accomplished by using rules such as:

- $if\ e_1\ then\ \sum\{\!|e_2 \mid x \in R|\!\}\ else\ e_3 \rightsquigarrow \sum\{\!|\ if\ e_1\ then\ e_2\ else\ e_3 \div N \mid x \in R|\!\}$
- $if\ e_1\ then\ e_2\ else\ \sum\{\!|e_3 \mid x \in R|\!\} \rightsquigarrow \sum\{\!|\ if\ e_1\ then\ e_2 \div N\ else\ e_3 \mid x \in R|\!\}$
- $e_1 \cdot \sum\{\!|e_2 \mid x \in R|\!\} \rightsquigarrow \sum\{\!|e_1 \cdot e_2 \mid x \in R|\!\}$
- $\sum\{\!|e_1 \mid x \in R|\!\} \cdot e_2 \rightsquigarrow \sum\{\!|e_1 \cdot e_2 \mid x \in R|\!\}$
- $\sum\{\!|e_1 \mid x \in R|\!\} \div e_2 \rightsquigarrow \sum\{\!|e_1 \div e_2 \mid x \in R|\!\}$
- $\sum\{\!|e_1 \mid x \in R|\!\} + e_2 \rightsquigarrow \sum\{\!|e_1 + (e_2 \div N) \mid x \in R|\!\}$
- $\sum\{\!|e_1 \mid x \in R|\!\} - e_2 \rightsquigarrow \sum\{\!|e_1 - (e_2 \div N) \mid x \in R|\!\}$
- $e_1 - \sum\{\!|e_2 \mid x \in R|\!\} \rightsquigarrow \sum\{\!|(e_1 \div N) - e_2 \mid x \in R|\!\}$
- $e_1 + \sum\{\!|e_2 \mid x \in R|\!\} \rightsquigarrow \sum\{\!|(e_1 \div N) + e_2 \mid x \in R|\!\}$
- $\sum\{\!|if\ e_1\ then\ e_2\ else\ e_3 \mid x \in R|\!\} \rightsquigarrow \sum\{\!|if\ e_1\ then\ e_2\ else\ 0 \mid x \in R|\!\} + \sum\{\!|if\ \neg e_1\ then\ e_3\ else\ 0 \mid x \in R|\!\}$, if neither e_2 nor e_3 is 0.

Having obtained F in this special form, the proof is continued by repeating the following steps until all occurrences of R have been eliminated.

Step 1. Look for an innermost subexpression of F that has the special form required by Proposition 4.3.2. Let this subexpression be F' and its free variables be y_1, ..., y_m, R and N. Generate all possible d-states with respect to these free variables of F'. The d is the smallest one suggested by Proposition 4.3.2 and serves as a lower bound for k. Let S_1, ..., S_{h+1} be these d-states. Apply Proposition 4.3.2 to F' with respect to each S_i to obtain expressions r_i which are ratios of polynomials of N. Then F' is equivalent to *if* S_1 *then* r_1 *else* ... *if* S_h *then* r_h *else* r_{h+1} under the assumption of the theorem that the variable R is never instantiated to short k'-multi-cycles where $k' < k$.

Step 2. To maintain the same special form, we need to push the S_i up one level to the expression in which F' is nested. This rewriting is done using rules such as:

- (*if* S_1 *then* r_1 ... *if* S_h *then* r_h *else* r_{h+1}) $=^{\mathbb{Q}} V \rightsquigarrow (S_1 \wedge r_1 =^{\mathbb{Q}} V) \vee \cdots \vee (S_{h+1} \wedge r_{h+1} =^{\mathbb{Q}} V)$
- *if* P *then* $(f \div$ (*if* S_1 *then* r_1 *else* ... *if* S_h *then* r_h *else* r_{h+1})) *else* e
  $\rightsquigarrow$ *if* $P \wedge S_1$ *then* $f \div r_1$ ... *if* $P \wedge S_{h+1}$ *then* $f \div r_{h+1}$ *else* e

Step 3. After Step 2, some expression having the form $U =^{\mathbb{Q}} V$, $U \leq V$, or their negation, can become an equation of ratios of polynomials of N. Such an expression can be replaced either by *true* or by *false*. For illustration, we explain the case of $U =^{\mathbb{Q}} V$; the other cases are similar. First, $U =^{\mathbb{Q}} V$ is readily transformed into a polynomial $P = 0$ with N being its only free variable. Check if P is identically 0. In that case, replace $U =^{\mathbb{Q}} V$ by *true*. If P is not identically 0, we use the fact that a polynomial has a finite number of roots. By choosing a sufficiently large lower bound for k, we can ensure that N always exceeds the largest root of P. Thus, in this case we replace $U =^{\mathbb{Q}} V$ by *false*.

Observe that in step 1 we have reduced the number of summations and in step 3 we have reduced the number of equality and inequality tests. By repeating these steps, we eventually reach the base case and arrive at an expression where R does not occur. When we are

finished, the resultant expression is clearly a boolean formula containing no free variable. Therefore its value does not depend on R. Consequently the theorem holds for any k not smaller than the lower bound determined by the above process. □

This theorem expresses a finite-cofiniteness of k-multi-cycle queries in the following sense. Let isomorphic k-multi-cycles be identified. Then for any $m \geq 1$, properties of k-multi-cycles consisting of at most m components are either finite or cofinite. This result is pregnant with implications. I present some of the obvious ones below.

Corollary 4.3.4 *Let* $chain : \{b \times b\} \rightarrow \mathbb{B}$ *be the predicate such that* $chain(O)$ *holds if and only if* O *is a chain. Then chain is not definable in* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$.

Proof. Let $singlecycle : \{b \times b\} \rightarrow \mathbb{B}$ be the predicate for testing if a graph is a single cycle. It is clear that for a k-multi-cycle O, $singlecycle(O)$ if and only if $chain(O - \{o\})$ for any $o \in O$. If *chain* is definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$, then the right-hand-side is definable in it too. This implies *singlecycle* is definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$, contradicting Theorem 4.3.3. □

Corollary 4.3.5 *Let* $connected : \{b \times b\} \rightarrow \mathbb{B}$ *be the predicate such that* $connected(O)$ *holds if and only if* O *is a connected graph. Then connected is not definable in* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$.

Proof. A k-multi-cycle O is connected if and only if it is a single cycle. Since $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ cannot test the latter, it cannot test the former. Note that this result holds for both directed connectivity and undirected connectivity. □

Corollary 4.3.6 *Let* $evencard : \{b \times b\} \rightarrow \mathbb{B}$ *be the predicate such that* $evencard(O)$ *holds if and only if* O *has even cardinality. Then evencard is not definable in* $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$.

Proof. By Theorem 4.3.3, there is no query in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ that can distinguish one k-multi-cycle from another as long as k is big enough. Therefore, there is

no query in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ that can distinguish a k-multi-cycle having an odd number of edges from a k-multi-cyle having an even number of edges, as long as k is big enough. The corollary follows immediately. □

Corollary 4.3.7 *Let $tc : \{b \times b\} \to \{b \times b\}$ be the function which computes the transitive closure of binary relations. Then tc is not definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$.*

Proof. Let $singlecycle : \{b \times b\} \to \mathbb{B}$ be a predicate such that $singlecycle(O)$ holds if and only if O is a graph having exactly one cycle. Clearly, $singlecycle(O)$ if and only if $tc(O) = cartprod(\Pi_1 O,\ \Pi_2 O)$. Hence definability of tc in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ implies definability of $singlecycle$ in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. By Theorem 4.3.3, $singlecycle$ is not definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. Hence neither is tc. □

A result similar to Corollary 4.3.7 was also obtained by Consens and Mendelzon [47]. They proved that if LOGSPACE is strictly included in NLOGSPACE, then transitive closure cannot be expressed in first-order logic augmented with certain aggregate functions. The separation of these two complexity classes has been and is likely to remain a difficult open problem. In contrast, my result does not require such a precondition. I should also point out that there is no simple alternative proof using complexity arguments of my corollaries above. It is known that many queries mentioned in the corollaries above are not in a low complexity class such as AC^0; see Johnson [111] and Furst, Saxe, and Sipser [69]. Hence if $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ can be shown to be in such a class, then many of my results would be immediate. However, $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ has higher complexity than AC^0; it has multiplication and it can test the parity of the cardinality of ordered sets. Neither of these capabilities belong to AC^0.

Expressible properties of k-strict-binary-trees are finite-cofinite

The proof of Theorem 4.3.3 relies on two things: satisfiability of d-states is easy to decide for k-multi-cycles and probabilities are easy to calculate and express as ratios of polynomials

in terms of the size of graphs for k-multi-cycles. There is another class of graphs having these two properties: k-strict-binary-trees. A k-strict-binary-tree is a nonempty tree where each node has either 0 or 2 decendents and the distance from the root to any leaf is at least k.

Theorem 4.3.8 *Let $G : \{b \times b\} \to \mathbb{B}$ be a function that is expressible in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. Then there is some k such that for all k-strict-binary-trees O, it is the case that $G(O)$ is true; or for all k-strict-binary-trees O, it is the case that $G(O)$ is false.*

Proof sketch. It is easy to decide if a d-state is satisfiable by some k-strict-binary-trees. The probability calculation is also simple. The only problem is that the probability must be expressed wholely as a ratio of polynomials of the number of edges in the tree. This is dealt with by observing that in k-strict-binary-trees, the number of internal nodes is 1 fewer than half the number of edges and the number of leaves is equal to 2 plus the number of internal nodes. The theorem follows by repeating verbatim the proof for k-multi-cycles. □

Therefore, no queries in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ can tell the difference of one k-strict-binary tree from another, provided k is big enough. It follows immediately that

Corollary 4.3.9 *Let balanced $: \{b \times b\} \to \mathbb{B}$ be a predicate such that balanced(O) holds if and only if O is a balanced binary tree. Then balanced is not definable in $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$.* □

Libkin and I [135] introduced a query language for bags by interpreting the syntax of $\mathcal{NRC}$ bag-theoretically. This bag language is equivalent to a sublanguage of $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$. It is also equivalent to the bag language of Grumbach and Milo [77] minus their power operators on bags. This equivalence allows us to use the results above to settled several conjectures on this bag query language. First is the conjecture of Grumbach and Milo [77] that this bag query language cannot test the parity of the cardinality of relations. This conjecture is implied by Corollary 4.3.6. Second is the conjecture of Grumbach and Milo [77] that this bag query language cannot define the transitive closure of relations. This

conjecture is implied by Corollary 4.3.7. Third and last is the conjecture of Paredaens [158] that this bag query language cannot test whether a binary tree is balanced or not. This conjecture is implied by Corollary 4.3.9.

Chapter 5

A Collection Programming Language called CPL

Based on some of the ideas described earlier on, I have built a prototype query system called Kleisli. (See Chapter 9.) The system is designed as a database engine to be connected to the host programming language ML [144] via a collection of libraries of routines. These routines are parameterized for more open and better control so that expert users do not have to resort to wily evasion of restriction in their quest for performance. (There is strong evidence [162] that experts demonstrate a canny persistence in uncovering necessary detail to satisfy their concern for performance.)

I have included an implementation of a high-level query language, for non-expert users, called CPL with the prototype. The libraries actually contain enough tools for a competent user to quickly build his own query language or command line interpreter to use in connection with Kleisli and the host language ML. In fact, CPL is an example of how to use these tools to implement query languages for Kleisli. This chapter is intended as an informal but accurate description of CPL.

Organization

Section 5.1. A rich data model is supported in CPL. In particular, sets, lists, bags, records, and variants can be freely combined. The language itself is obtained by orthogonally combining constructs for manipulating these data types. The data types and the core fragment of CPL is described in this section. Examples are provided to illustrate CPL's modeling power.

Section 5.2. A comprehension syntax is used in CPL to uniformly manipulate sets, lists, and bags. CPL's comprehension notation is a generalization of the list comprehension notation of functional languages such as Miranda [188]. The comprehension syntax of CPL is presented in this section and its semantics is explained in terms of core CPL. Examples are provided to illustrate the uniform nature of list-, bag-, and set-comprehensions.

Section 5.3. A pattern matching mechanism is supported in CPL. In particular, convenient partial-record patterns and variable-as-constant patterns are supported. The former is also available in languages such as Machiavelli [153], but not in languages such as ML [144]. The latter feature is not available elsewhere at all. The pattern matching mechanism of CPL is presented in two stages. In the first stage, simple patterns are described. In the second stage, enhanced patterns are described. Semantics is again given in terms of core CPL. Examples are provided to illustrate the convenience of pattern matching.

Section 5.4. More examples are given to illustrate other features of CPL. These features include: (1) Types are automatically inferred in CPL. In particular, CPL has polymorphic record types. However, the type inference system is simpler than that of Ohori [154], Remy [165], etc. (2) External functions can be easily imported from the host system into CPL. Scanners and writers for external data can be easily added to CPL. More details can be found in Chapter 9. (3) An extensible optimizer is available. The basic optimizer does loop fusion, filter promotion, and code motion. It optimizes scanning and printing of external files. It has been extended to deal with joins by picking alternative join operators and by migrating them to external servers. More details can be found in Chapters 6 and 7.

5.1 The core of CPL

I first describe CPL's types. Then I describe the core fragment of CPL. The core fragment is based on the central idea of restricting structural recursion to homomorphisms of collection types. In fact, when restricted to sets, CPL is really a heavily sugared version of $\mathcal{NRC}(=)$. Lastly, several examples are provided to illustrate CPL's modeling power.

Types

The ground complex object types, ranged over by s and t, are given by the grammar below. The l_i are labels and are required to be distinct. The b ranges over base types.

$$s ::= b \mid \{s\} \mid \{|s|\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : s_1, \cdots, l_n : s_n\texttt{)} \mid \texttt{<}l_1 : s_1, \cdots, l_n : s_n\texttt{>}$$

The ground types, over which u and v range, are given by the grammar below.

$$u ::= b \mid \{s\} \mid \{|s|\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)} \mid \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \mid u \texttt{ -> } v$$

CPL allows $\texttt{(}u_1, \cdots, u_n\texttt{)}$ as a syntactic sugar for $\texttt{(}\#1 : u_1, \cdots, \#n : u_n\texttt{)}$. Labels in CPL always start with the #-sign.

Objects of type $\{s\}$ are finite sets whose elements are objects of type s. Objects of type $\{|s|\}$ are finite bags whose elements are objects of type s. Objects of type $\texttt{[}s\texttt{]}$ are finite lists whose elements are objects of type s. Objects of type $\texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)}$ are records having exactly fields l_1, ..., l_n and whose values at these fields are objects of types u_1, ..., u_n respectively. An object of type $\texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>}$ is called a variant object and is a pair $\texttt{<}l_i : o\texttt{>}$ such that o is an object of type u_i; that is, it is an object of type u_i tagged with the label l_i. (Variants are also called tagged-unions and co-products. See Gunter [82] or Hull and Yap [100] for more information.) Objects of type u `->` v are functions from type u to type v. Included amongst the base types are `int`, `real`, `string`, `bool`, and `unit` (which is the type having exactly the empty record `()` as its only object).

Observe that function types u `->` v in CPL are higher-order because u and v can contain function types. This is different from the first-order function types $s \to t$ of $\mathcal{NRC}$ in the previous chapters. Higher-order function types are allowed in CPL for two reasons. To discuss these two reasons, I need some results obtained by Suciu and myself [180] on the forms of structural recursion sri and sru. Let $\mathcal{HNRC}(\mathbb{B}, =, sri)$ and $\mathcal{HNRC}(\mathbb{B}, =, sru)$ respectively denote the language obtained by generalizing $\mathcal{NRC}(\mathbb{B}, =, sri)$ and $\mathcal{NRC}(\mathbb{B}, =, sru)$ to higher-order function types. We showed that $\mathcal{HNRC}(\mathbb{B}, =, sri) = \mathcal{HNRC}(\mathbb{B}, =, sru) = \mathcal{NRC}(\mathbb{B}, =, sri) = \mathcal{NRC}(\mathbb{B}, =, sru)$ over the class of first-order functions. Hence every function of type $s \to t$ expressible using the higher-order languages is also expressible using the first-order languages. This result gives us the first reason for having higher-order function types in CPL— it makes many things more convenient but without making analysis of first-order expressive power of CPL more complicated. Another result in the same paper [180] is that all uniform implementations of sri in $\mathcal{NRC}(\mathbb{B}, =, sru)$ are bound to be expensive while there are efficient uniform implementations of sri in $\mathcal{HNRC}(\mathbb{B}, =, sru)$. This gives us the second reason for having higher-order function types in CPL— it allows the implementation of more efficient algorithms. See my paper with Suciu [180] for details.

CPL also has nonground types. I only intend to explain how to read a CPL type expression having nonground types below. The nonground complex object types are ranged over by the symbol σ. Nonground complex object types are obtained from ground complex object types by replacing some subexpressions with complex object type variables of the following forms:

- Unconstrained complex object type variable. It has the form `''n`, where n is a natural number. It can be instantiated to any ground complex object type.
- Record complex object type variable. It has the form $(l_1 : \sigma_1, \cdots, l_n : \sigma_n;$ `''m`$)$, where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.
- Variant complex object type variable. It has the form $\langle l_1 : \sigma_1, \cdots, l_n : \sigma_n;$ `''m`$\rangle$, where

m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.

The nonground types are ranged over by the symbol β. Nonground types are obtained from ground types by replacing some subexpressions with a universal type variable of the following forms:

- Unconstrained universal type variable. It has the form `'n`, where n is a natural number. It can be instantiated to any ground type.
- Record universal type variable. It has the form $(l_1 : \beta_1, \cdots, l_n : \beta_n;$ `'m`$)$, where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.
- Variant universal type variable. It has the form $\langle l_1 : \beta_1, \cdots, l_n : \beta_n;$ `'m>`, where m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.

Hence for example `(#age : string; '1) -> string` indicates the type of functions whose inputs are records having at least the field `#age` of `string` type and producing outputs of `string` type. Similarly, `(#age : string; '1) -> (#age : string; '1)` indicates the type of functions whose inputs are records having at least the fields `#age` of `string` type and produces outputs of exactly the same type as the inputs.

The distinction between nonground complex object types and nonground types is that the latter types include function types but the former types do not. For example, the nonground complex object type `(#input_set: {int}; ''1)` cannot be instantiated to a record type such as `(#input_set: {int}, #transformer: int->int)`. On the other hand, the nonground type `(#input_set: {int}; '1)` can be instantiated to a record type such as `(#input_set: {int}, #transformer: int->int)`.

CPL also has a token stream type `[|`s`|]`. An object of type `[|`s`|]` is a token stream

representing an object of type s. As token streams seldom appear in normal user programs, I omit them from this description of CPL.

Expressions

The expressions are ranged over by e. The variables are ranged over by x. For simplicity, we assign a ground type once-and-forever to all the variables; the type u assigned to a variable x is indicated by superscripting: x^u. Expression formation constructs are based on the structure of types. See also the comments at end of Section 2.1.

For function types, the expression constructs are given in Figure 5.1. The meaning of $\backslash x^u$ `=>` e is the function f that when applied to an object o of type u produces the object $e[o/x^u]$. (The notation $e[o/x^u]$ means replace all free occurrences of x^u in e by o.) The meaning of e_1 `@` e_2 is the result of applying the function e_1 to the object e_2.

$$\frac{}{x^u : u} \qquad \frac{e : v}{\backslash x^u \texttt{ => } e : u \texttt{ -> } v} \qquad \frac{e_1 : u \texttt{ -> } v \quad e_2 : u}{e_1 \texttt{ @ } e_2 : v}$$

Figure 5.1: The constructs for function types in CPL.

For record types, the expression constructs are given in Figure 5.2. I have already mentioned that `()` is the unique object having type `unit`. The construct $(l_1 : e_1, \cdots, l_n : e_n)$ forms a record having fields l_1, ..., l_n whose values are e_1, ..., e_n respectively. A label when used as an expression stands for the obvious projection function.

For variant types, the expression constructs are given in Figure 5.3. The construct `<`l : e`>` forms a variant object by tagging the object e with the label l. The case-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`l_i : o`>`. The case-otherwise-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`l_i : o`>` where $1 \leq i \leq n$; otherwise it evaluates to e'. See also Section 2.3.

$$\frac{}{() : \texttt{unit}} \qquad \frac{e_1 : u_1 \quad \cdots \quad e_n : u_n}{(l_1 : e_1, \cdots, l_n : e_n) : (l_1 : u_1, \cdots, l_n : u_n)}$$

$$\frac{}{l^{(l:u,\ l_1:u_1,\cdots,l_n:u_n)} : (l : u,\ l_1 : u_1, \cdots, l_n : u_n) \texttt{ -> } u}$$

Figure 5.2: The constructs for record types in CPL.

$$\frac{e : u}{\texttt{<}l : e\texttt{>}^{\texttt{<}l_1:u_1,\cdots,l_n:u_n\texttt{>}} : \texttt{<}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{>}}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u}{\texttt{case } e \texttt{ of } \texttt{<}l_1 : \backslash x_1^{u_1}\texttt{> => } e_1 \texttt{ or } \cdots \texttt{ or } \texttt{<}l_n : \backslash x_n^{u_n}\texttt{> => } e_n : u}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_{n+m} : u_{n+m}\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u \quad e' : u}{\texttt{case } e \texttt{ of } \texttt{<}l_1 : \backslash x_1^{u_1}\texttt{> => } e_1 \texttt{ or } \cdots \texttt{ or } \texttt{<}l_n : \backslash x_n^{u_n}\texttt{> => } e_n \texttt{ otherwise } e' : u}$$

Figure 5.3: The constructs for variant types in CPL.

For the base type `bool`, there are the usual constructs given in Figure 5.4.

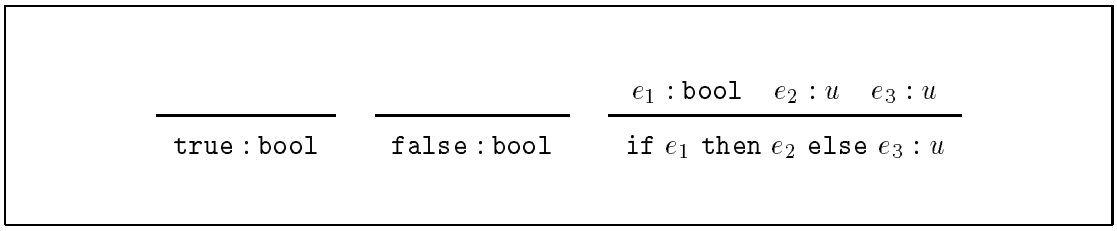


Figure 5.4: The constructs for the Boolean type in CPL.

For set types, the expression constructs are given in Figure 5.5. The meaning of $\{\}^s$ is the empty set. The meaning of $\{e\}$ is the singleton set containing e. The meaning of e_1 {+} e_2 is the set union of e_1 and e_2. The expression sext$\{e_1 \mid \backslash x^t$ <- $e_2\}$ stands for the set $e_1[o_1/x^t]$ {+} $\cdots$ {+} $e_1[o_n/x^t]$, where o_1, ..., o_n are all the elements of the set e_2.

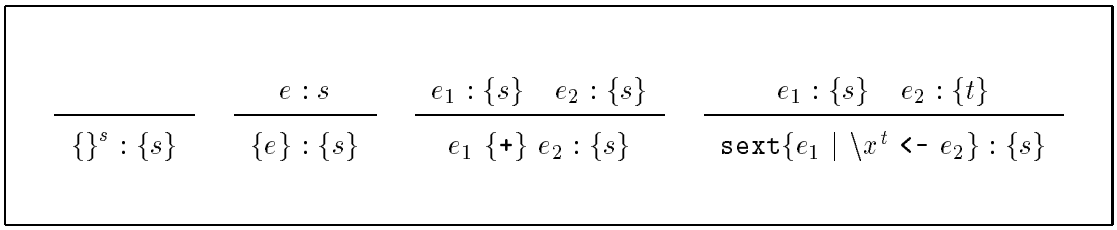


Figure 5.5: The constructs for set types in CPL.

For bag types, the expression constructs are given in Figure 5.6. The meaning of $\{|\,|\}^s$ is the empty bag. The meaning of $\{|e|\}$ is the singleton bag containing e. The meaning of e_1 $\{|+|\}$ e_2 is the bag union of e_1 and e_2; it is sometimes called the additive union. For example, if e_1 is a bag of five apples and two oranges and e_2 is a bag of one apple and three oranges, then e_1 $\{|+|\}$ e_2 is a bag of six apples and five oranges. The expression bext$\{|e_1 \mid \backslash x^t$ <-- $e_2|\}$ stands for the bag $e_1[o_1/x^t]$ $\{|+|\}$ $\cdots$ $\{|+|\}$ $e_1[o_n/x^t]$. More information on bags can be found in [130].

For list types, the expression constructs are given in Figure 5.7. The meaning of $[]^s$ is the empty list. The meaning of [e] is the singleton list containing e. The meaning of e_1 [+] e_2

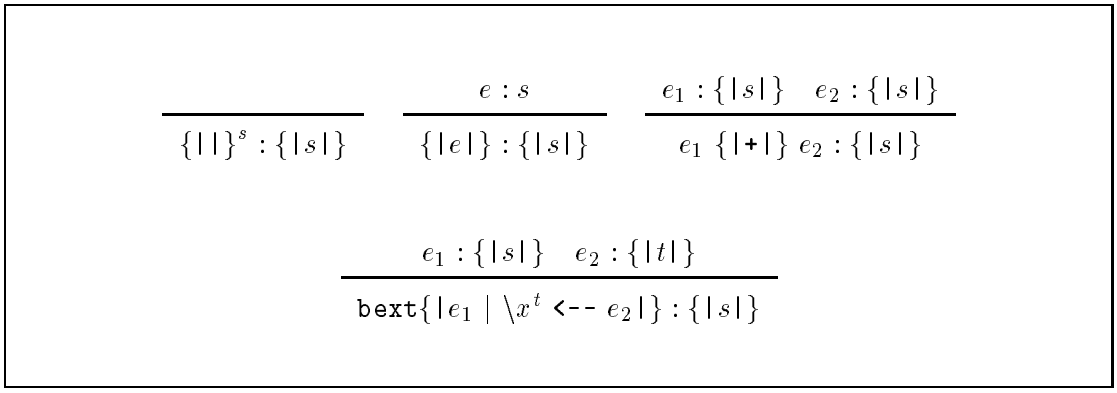


Figure 5.6: The constructs for bag types in CPL.

is the list concatenation of e_1 and e_2. The expression `lext[`e_1 | $\backslash x^t$ `<---` e_2`]` stands for the list $e_1[o_1/x^t]$ `[+]` $\cdots$ `[+]` $e_1[o_n/x^t]$.

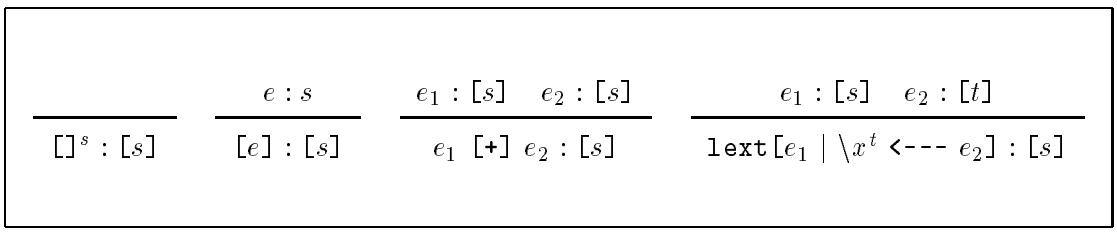
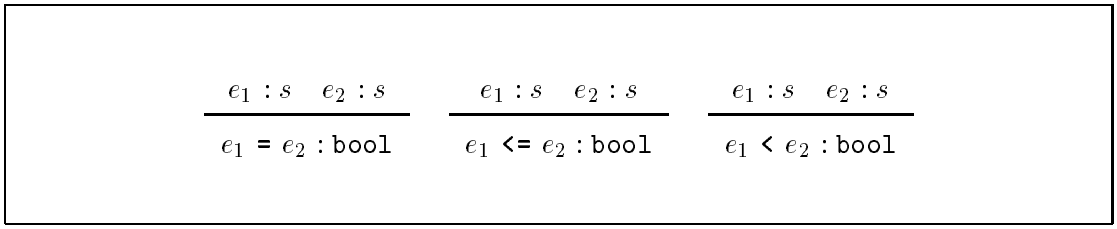


Figure 5.7: The constructs for list types in CPL.

CPL also includes the primitives functions listed in Figure 5.8 for comparing complex objects. The operator `=` is the equality test. The operator `<=` is the linear order. The operator `<` is the strict version. The linear order is based on the technique of lifting presented in Section 3.2.

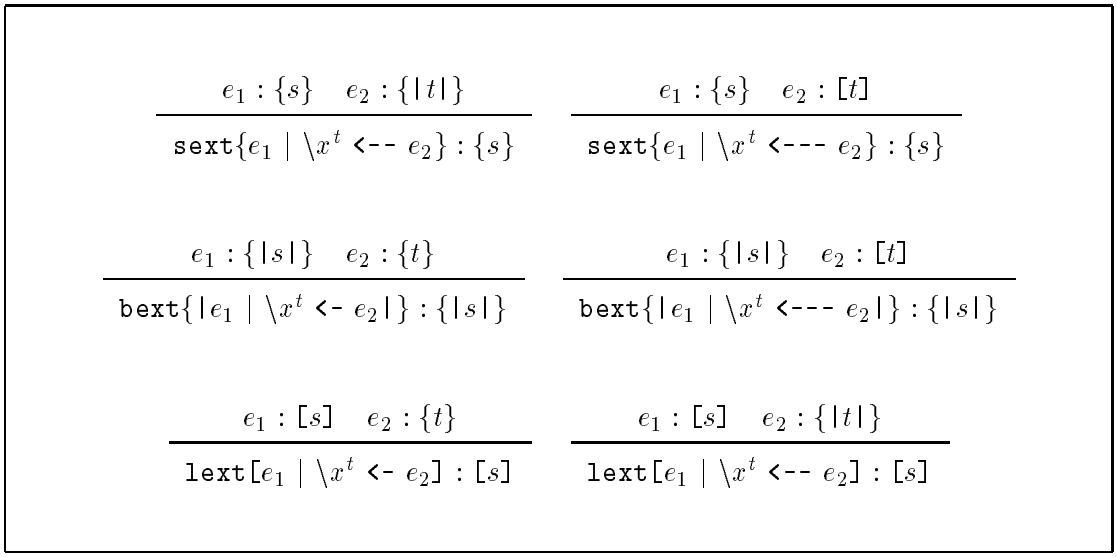
CPL supports conversion between lists, bags, and sets. These operators, listed in Figure 5.9, correspond to the monad morphisms mentioned in Wadler [198]. The expression `sext{`e_1 | $\backslash x^t$ `<--` e_2`}` stands for the set $e_1[o_1/x^t]$ `{+}` $\cdots$ `{+}` $e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the bag e_2. The expression `bext{|`e_1 | $\backslash x^t$ `<-` e_2`|}` stands for the bag $e_1[o_1/x^t]$ `{|+|}` $\cdots$ `{|+|}` $e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the set e_2. The expression `sext{`e_1 | $\backslash x^t$ `<---` e_2`}` stands for the set $e_1[o_1/x^t]$ `{+}` $\cdots$ `{+}` $e_1[o_n/x^t]$,



$$\frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ = } e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ <= } e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ < } e_2 : \texttt{bool}}$$

Figure 5.8: The constructs for comparing objects in CPL.

where o_1, ..., o_n are the distinct elements in the list e_2. The expression $\texttt{lext[}e_1 \mid \backslash x^t \texttt{ <- } e_2\texttt{]}$ stands for the list $e_1[o_1/x^t]$ `[+]` $\cdots$ `[+]` $e_1[o_n/x^t]$, where o_1, ..., o_n are the distinct elements in the set e_2 and o_1 `<` $\cdots$ `<` o_n. The expression $\texttt{bext\{|}e_1 \mid \backslash x^t \texttt{ <--- } e_2\texttt{|\}}$ stands for the bag $e_1[o_1/x^t]$ $\{\texttt{|+|}\}$ $\cdots$ $\{\texttt{|+|}\}$ $e_1[o_n/x^t]$, where o_1, ..., o_n are the elements in the list e_2, with o_i occurring at position i. The expression $\texttt{lext[}e_1 \mid \backslash x^t \texttt{ <-- } e_2\texttt{]}$ stands for the list $e_1[o_1/x^t]$ `[+]` $\cdots$ `[+]` $e_1[o_n/x^t]$, where o_1, ..., o_n are the elements in the bag e_2 and o_1 `<=` $\cdots$ `<=` o_n.



$$\frac{e_1 : \{s\} \quad e_2 : \{|\,t\,|\}}{\texttt{sext}\{e_1 \mid \backslash x^t \texttt{ <-- } e_2\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : [t]}{\texttt{sext}\{e_1 \mid \backslash x^t \texttt{ <--- } e_2\} : \{s\}}$$

$$\frac{e_1 : \{|\,s\,|\} \quad e_2 : \{t\}}{\texttt{bext}\{|\,e_1 \mid \backslash x^t \texttt{ <- } e_2\,|\} : \{|\,s\,|\}} \qquad \frac{e_1 : \{|\,s\,|\} \quad e_2 : [t]}{\texttt{bext}\{|\,e_1 \mid \backslash x^t \texttt{ <--- } e_2\,|\} : \{|\,s\,|\}}$$

$$\frac{e_1 : [s] \quad e_2 : \{t\}}{\texttt{lext[}e_1 \mid \backslash x^t \texttt{ <- } e_2\texttt{]} : [s]} \qquad \frac{e_1 : [s] \quad e_2 : \{|\,t\,|\}}{\texttt{lext[}e_1 \mid \backslash x^t \texttt{ <-- } e_2\texttt{]} : [s]}$$

Figure 5.9: The constructs for list-bag-set interactions in CPL.

CPL supports some syntactic sugar on expressions:

- The expression e_1 `.` e_2 means e_2 `@` e_1.
- The expression $e_1\ e_2\ e_3$ is the binary function application in infix form for e_2 `@` (e_1, e_2); all binary functions in CPL can be applied in infix form.
- The expression e_1 `o` e_2 means $\backslash x^u$ `=>` e_1 `@` $(e_2$ `@` $x^u)$ where x^u is fresh and u is the appropriate type.
- The expression $(e_1, \cdots, e_n)$ means $(\#1 : e_1, \cdots, \#n : e_n)$.
- The expression `let` $\backslash x^s == e_1$ `in` e_2 means $(\backslash x^s$ `=>` $e_2)$ `@` e_1.
- The expression $\{e_1, \cdots, e_n\}$ means $\{e_1\}$ `{+}` $\cdots$ `{+}` $\{e_n\}$.
- The expression $\{|e_1, \cdots, e_n|\}$ means $\{|e_1|\}$ `{|+|}` $\cdots$ `{|+|}` $\{|e_n|\}$.
- The expression `[`$e_1, \cdots, e_n$`]` means `[`e_1`]` `[+]` $\cdots$ `[+]` `[`e_n`]`.
- The expression e_1 `+}` e_2 means $\{e_1\}$ `{+}` e_2.
- The expression e_1 `+|}` e_2 means $\{|e_1|\}$ `{|+|}` e_2.
- The expression e_1 `+]` e_2 means `[`e_1`]` `[+]` e_2.

CPL comes with a type inference system that is considerably simpler than those of Ohori [154], Remy [165], etc., because CPL does not have a record-concatenation operation. Hence there is no need to indicate types any where in CPL expressions. So we drop our type superscripts henceforth, except when giving typing rules.

Examples: CPL's modeling power

Sets, lists, bags [130], records, and variants [82] are supported in CPL. These types can be freely combined, giving rise to a rich and flexible data model.

Example. Here is a list of sets of numbers in CPL:

```
[ { 1, 2, 3}, {4, 5, 6}, {}, {3, 7} ] ;
Result : [{1, 2, 3}, {4, 5, 6}, {}, {3, 7}]
Type   : [{int}]
```

Example. We can model the employee salary history example of Makinouchi [141] by a nested relation as below.

```
{(#name: "tom", #history: {(#date: "june 1993", #salary: 2000),
(#date: "july 1993", #salary: 2100)}), (#name: "jim", #history: {})};
Result : {(#history: {},
           #name: "jim"),
          (#history: {(#salary: 2100,
                       #date: "july 1993"),
                      (#salary: 2000,
                       #date: "june 1993")},
           #name: "tom")}
Type   : {(#history:{(#salary:int, #date:string)}, #name:string)}
```

Notice that "jim" has the empty set as his salary history; he is probably a new employee. Had we not used nested relations, we must resort to either two flat tables (one for new employees and one for old employees) or to null values.

Example. We model student information, where some of them have phone number as contact address and some have room number instead. This is done using variants:

```
{(#name:"jim", #contact:<#phone:"3-4560">), (#name:"tom", #contact:
<#office:"2-2210">)} ;
Result : {(#contact: <#office: "2-2210">,
           #name: "tom"),
          (#contact: <#phone: "3-4560">,
           #name: "jim")}
```

```
Type : {(#contact:<#office:string,#phone:string;''2>,#name:string)}
```

Had we not used variants, we must resort to either two flat tables (one for people having phone number and one for those who have room number) or to null values.

5.2 Collection comprehension in CPL

An important influence on the design of CPL is Wadler's idea of using the comprehension syntax for manipulating monads [198]. His idea is to introduce a comprehension construct $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ in place of the $\bigcup\{e_1 \mid x \in e_2\}$ construct of $\mathcal{NRC}$. This construct can be interpreted in $\mathcal{NRC}$ by treating $\{e \mid x \in e', \Delta\}$ as $\bigcup\{\{e \mid \Delta\} \mid x \in e'\}$ and $\{e \mid \}$ as $\{e\}$. Conversely the $\bigcup\{e_1 \mid x \in e_2\}$ construct can be interpreted as $\{y \mid x \in e_2,\ y \in e_1\}$ in Wadler's language. Thus his language is equivalent to $\mathcal{NRC}$.

The comprehension syntax is less abstract than $\mathcal{NRC}$ for the purpose of theoretical study. However, it is very appealing for the purpose of everyday programming. Therefore, I have added a collection comprehension mechanism to CPL. This mechanism is similar to the list comprehension mechanism in functional languages such as KRC [187] and Haskell [62]. However, CPL's version is slightly more general.

Collection comprehensions in CPL

There are three constructs for collection comprehension, one each for sets, bags, and lists. The typing rules are given in Figure 5.10, where A_i and A_i* has one of the following forms:

- A_i is an expression e_i. Then A_i* is the type-derivation showing e_i : `bool`.
- A_i a set-abstraction $\backslash x^{s_i}$ `<-` e_i. Then A_i* is the type-derivation showing $e_i : \{s_i\}$.
- A_i is a bag-abstraction $\backslash x^{s_i}$ `<--` e_i. Then A_i* is the type-derivation showing e_i : $\{\mid s_i \mid\}$.

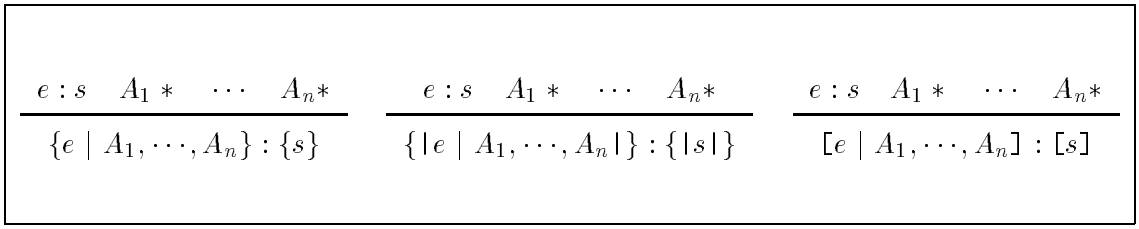


Figure 5.10: The comprehension constructs in CPL.

- A_i is a list-abstraction $\backslash x^{s_i}$ `<---` e_i. Then A_i* is the type-derivation showing e_i : `[`s_i`]`.

I now define the semantics of these comprehension constructs in terms of the various `ext` constructs introduced earlier. The translation used is based on that suggested by Wadler [198]. Let us use Δ as a meta notation for a sequence of A_i.

For set comprehensions:

- Interpret $\{e' \mid \backslash x^s$ `<-` $e, \Delta\}$ as `sext`$\{\{e' \mid \Delta\} \mid \backslash x^s$ `<-` $e\}$.
- Interpret $\{e' \mid \backslash x^s$ `<--` $e, \Delta\}$ as `sext`$\{\{e' \mid \Delta\} \mid \backslash x^s$ `<--` $e\}$.
- Interpret $\{e' \mid \backslash x^s$ `<---` $e, \Delta\}$ as `sext`$\{\{e' \mid \Delta\} \mid \backslash x^s$ `<---` $e\}$.
- Interpret $\{e' \mid e, \Delta\}$ as `if` e `then` $\{e' \mid \Delta\}$ `else` $\{\}$.

For bag comprehensions:

- Interpret $\{\mid e' \mid \backslash x^s$ `<-` $e, \Delta \mid\}$ as `bext`$\{\mid\{\mid e' \mid \Delta \mid\} \mid \backslash x^s$ `<-` $e \mid\}$.
- Interpret $\{\mid e' \mid \backslash x^s$ `<--` $e, \Delta \mid\}$ as `bext`$\{\mid\{\mid e' \mid \Delta \mid\} \mid \backslash x^s$ `<--` $e \mid\}$.
- Interpret $\{\mid e' \mid \backslash x^s$ `<---` $e, \Delta \mid\}$ as `bext`$\{\mid\{\mid e' \mid \Delta \mid\} \mid \backslash x^s$ `<---` $e \mid\}$.
- Interpret $\{\mid e' \mid e, \Delta \mid\}$ as `if` e `then` $\{\mid e' \mid \Delta \mid\}$ `else` $\{\mid\mid\}$.

For list comprehensions:

- Interpret `[`e' `|` $\backslash x^s$ `<-` e, Δ`]` as `lext[[`e' `|` Δ`]` `|` $\backslash x^s$ `<-` e`]`.
- Interpret `[`e' `|` $\backslash x^s$ `<--` e, Δ`]` as `lext[[`e' `|` Δ`]` `|` $\backslash x^s$ `<--` e`]`.
- Interpret `[`e' `|` $\backslash x^s$ `<---` e, Δ`]` as `lext[[`e' `|` Δ`]` `|` $\backslash x^s$ `<---` e`]`.
- Interpret `[`e' `|` e, Δ`]` as `if` e `then [`e' `|` Δ`] else []`.

The basic idea of interpreting comprehension in terms of the monad transformation constructs `ext` is due to Wadler [198]. Wadler explicitly considered the situation of $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ where e_1, ..., e_n come from the same monad (in this case set). He also had the idea of monad morphism that takes objects from one kind of monad to a different kind of monad. For some reason, he did not take the obvious step of building monad morphism into his comprehension syntax. My comprehension syntax directly incorporates the six special cases of monad morphism (set/bag/list-conversions) above.

Examples: Uniform collection manipulation with comprehension

Comprehension notations are used in CPL to uniformly manipulate sets, lists, and bags. This mechanism is a generalization of the list comprehension mechanism in functional languages like Haskell [96], Miranda [188], KRC [187], Id [152], etc. As demonstrated by Trinder [185], this is a rather natural notation for writing queries.

Example. The cartesian product on sets can be written in CPL as below. (Note: `primitive P == e` is CPL's syntax for explicitly naming a value.)

```
primitive cpSet == (\x, \y) => { (u, v) | \u <- x, \v <- y } ;
Result : Primitive cpSet registered.
Type   : (#1: {''1}, #2: {''2}) -> {(#1: ''1, #2: ''2)}

{1,2} cpSet {3,4} ;
Result : {(#1:2, #2:4), (#1:2, #2:3), (#1:1, #2:4), (#1:1, #2:3)}
Type   : {(#1:int, #2:int)}
```

Example. The cartesian product on lists can be written in CPL as follows, where set-brackets are replaced by list-brackets and set-abstractions are replaced by list-abstractions:

```
primitive cpList == (\x, \y) => [ (u, v) | \u <--- x, \v <--- y ] ;
Result : Primitive cpList registered.
Type   : (#1: [''1], #2: [''2]) -> [(#1: ''1, #2: ''2)]

["a", "b"] cpList ["c", "d"] ;
Result : [(#1: "a", #2: "c"), (#1: "a", #2: "d"),
          (#1: "b", #2: "c"), (#1: "b", #2: "d")]
Type   : [(#1:string, #2:string)]
```

Example. Conversion between lists, bags, and sets is very natural. Here is the function that selects all positive numbers in a list and puts them in a set.

```
primitive positiveListToSet == \x => { y | \y <--- x, 0 <= y } ;
Result : Primitive positiveListToSet registered.
Type   : [int]->{int}

positiveListToSet @ [~1, 2, ~3, 5] ;
Result : { 2, 5 }
Type   : {int}
```

5.3 Pattern matching in CPL

To further increase the user-friendliness of CPL queries, I add a pattern-matching mechanism to CPL. This mechanism is more general than that found in languages such as HOPE [34] and ML [144]. In particular, it supports partial-record patterns found in Machiavelli [153] and it supports variable-as-constant patterns not found anywhere else. I introduce the pattern matching mechanism in two stages, viz. simple patterns and enhanced patterns.

Simple patterns

I use the meta symbol S to range over simple patterns. The grammar is given below:

S	::=	`_`	Match anything
	|	$\backslash x$	Match anything and bind it to x
	|	$\backslash x$`&`S	Match using S and bind it to x
	|	$(l_1 : S_1, \cdots, l_n : S_n)$	Match records
	|	$(l_1 : S_1, \cdots, l_n : S_n, ...)$	Match records partially
	|	`()`	Match `()` only

A pattern must also satisfy the constraint that no $\backslash x$ is allowed to appear more than once in it. The last three dots in the pattern $(l_1 : S_1, \cdots, l_n : S_n, ...)$ are part of the syntax; this is called the partial record patten. Also I say a pattern is ultra-simple if it is just $\backslash x$. CPL also support the pattern $(S_1, \cdots, S_n)$ as syntactic sugar for the pattern $(\#1 : S_1, \cdots, \#n : S_n)$.

Simple patterns are used in lambda abstraction, case-expression, case-otherwise-expression, set abstraction, bag abstraction, and list abstraction. That is, they can be used anywhere a $\backslash x$ can be used. I now define the semantics of these patterns in terms of the core language presented earlier. The translation is given by cases below.

For lambda abstraction:

- Treat `_ =>` e as $\backslash x$ `=>` e where x is fresh.
- Treat $\backslash x$`&`S `=>` e as $\backslash x$ `=>` $(S$ `=>` $e)$ `@` x.
- Treat $(l_1 : S_1, \cdots, l_n : S_n)$ `=>` e as $\backslash x$ `=>` $(S_1$ `=>` $\cdots(S_n$ `=>` e `@` $(x\ .\ l_n))\cdots)$ `@` $(x\ .\ l_1)$
- Treat $(l_1 : S_1, \cdots, l_n : S_n, ...)$ `=>` e as $\backslash x$ `=>` $(S_1$ `=>` $\cdots(S_n$ `=>` e `@` $(x\ .\ l_n))\cdots)$ `@` $(x\ .\ l_1)$
- Treat `()` `=>` e as $\backslash x^{\texttt{unit}}$ `=>` e, where $x^{\texttt{unit}}$ is fresh.

For case-expressions:

- Treat `case` e `of <`l_1 `:` S_1`> =>` e_1 `or` $\cdots$ `or <`l_n `:` S_n`> =>` e_n as `case` e `of <`l_1 `:` $\backslash x_1$`> =>` $(S_1$ `=>` $e_1)$ `@` x_1 `or` $\cdots$ `or <`l_n `:` $\backslash x_n$`> =>` $(S_n$ `=>` $e_n)$ `@` x_n, where all x_i are fresh and some S_i are not ultra-simple.

- Treat `case` e `of <`l_1 `:` S_1`> =>` e_1 `or` $\cdots$ `or <`l_n `:` S_n`> =>` e_n `otherwise` e' as `case` e `of <` l_1 `:` $\backslash x_1$ `> =>` $(S_1$ `=>` $e_1)$ `@` x_1 `or` $\cdots$ `or <`l_n `:` $\backslash x_n$`> =>` $(S_n$ `=>` $e_n)$ `@` x_n `otherwise` e', where all x_i are fresh and some S_i are not ultra-simple.

For collection abstractions, I provide only the cases of `sext` for illustration. The cases for `bext` and `lext` are analogous.

- Treat `sext`$\{e_1$ | S `<-` $e_2\}$ as `sext`$\{(S$ `=>` $e_1)$ `@` x | $\backslash x$ `<-` $e_2\}$, where x is fresh and S is not ultra-simple.

- Treat `sext`$\{e_1$ | S `<--` $e_2\}$ as `sext`$\{(S$ `=>` $e_1)$ `@` x | $\backslash x$ `<--` $e_2\}$, where x is fresh and S is not ultra-simple.

- Treat `sext`$\{e_1$ | S `<---` $e_2\}$ as `sext`$\{(S$ `=>` $e_1)$ `@` x | $\backslash x$ `<---` $e_2\}$, where x is fresh and S is not ultra-simple.

Enhanced patterns

I use the meta symbol E to range over enhanced patterns. Enhanced patterns are a generalization of simple patterns. The grammar is given below:

E	::=	_	Match anything
	|	$\backslash x$	Match anything and bind it to x
	|	$\backslash x$`&`E	Match using E and bind it to x
	|	$(l_1 : E_1, \cdots, l_n : E_n)$	Match records
	|	$(l_1 : E_1, \cdots, l_n : E_n, ...)$	Match records partially
	|	`()`	Match `()` only
	|	c	Match constant c only
	|	`<`l : E`>`	Match variants
	|	x	Match the value bound to x only

Observe that in simple patterns every occurrence of a variable x is slashed, as in $\backslash x$. In enhanced patterns, a variable can appear without being slashed. A pattern where a variable x occurs without being slashed is called a variable-as-constant pattern. As before, a pattern must satisfy the constraint that no $\backslash x$ is allowed to appear more than once in it; however, unslashed variables can appear as frequently as desired.

Enhanced patterns are used only in set abstraction, bag abstraction, and list abstraction. I define their semantics in terms of simple patterns. I give the cases for `sext` for illustrations. The other cases are analogous.

- Treat `sext`$\{e_1$ | E `<-` $e_2\}$ as `sext`$\{$`if` $x = A$ `then` e_1 `else` $\{\}$ | E' `<-` $e_2\}$, where x is fresh, A is a subpattern in E and is either a constant or an unslashed variable, and E' is obtained from E by replacing one occurrence of A with $\backslash x$.
- Treat `sext`$\{e_1$ | E `<-` $e_2\}$ as `sext`$\{$`case` x *of* `<`l : $\backslash y$`> =>` `sext`$\{e_1$ | E'' `<-` $\{y\}\}$ `otherwise` $\{\}$ | E' `<-` $e_2\}$, where x and y are fresh, `<`l : E''`>` is a subpattern in E and E' is obtained from E by replacing one occurrence of `<`l : E''`>` by $\backslash x$.
- Treat `sext`$\{e_1$ | E `<--` $e_2\}$ as `sext`$\{$`if` $x = A$ `then` e_1 `else` $\{\}$ | E' `<--` $e_2\}$, where x

is fresh, A is a subpattern in E and is either a constant or an unslashed variable, and E' is obtained from E by replacing one occurrence of A with $\backslash x$.

- Treat `sext{`e_1 | E `<--` e_2`}` as `sext{case` x *of* `<`l : $\backslash y$`> => sext{`e_1 | E'' `<-` $\{y\}$`} otherwise {}` | E' `<--` e_2`}`, where x and y are fresh, `<`$l : E''$`>` is a subpattern in E and E' is obtained from E by replacing one occurrence of `<`$l : E''$`>` by $\backslash x$.

- Treat `sext{`e_1 | E `<---` e_2`}` as `sext{if` $x = A$ `then` e_1 `else {}` | E' `<---` e_2`}`, where x is fresh, A is a subpattern in E and is either a constant or an unslashed variable and E' is obtained from E by replacing one occurrence of A with $\backslash x$.

- Treat `sext{`e_1 | E `<---` e_2`}` as `sext{case` x *of* `<`l : $\backslash y$`> => sext{`e_1 | E'' `<-` $\{y\}$`} otherwise {}` | E' `<---` e_2`}`, where x and y are fresh, `<`$l : E''$`>` is a subpattern in E and E' is obtained from E by replacing one occurrence of `<`$l : E''$`>` by $\backslash x$.

This is a good place to explain the motivation of slashing a variable on its introduction. Consider the expression $\lambda x.\{(x, y) \mid (x, y) \in R\}$ written in a comprehension notation consistent with Wadler's [198]. On first sight, this seems to be a program which takes in an x and then selects from the relation R every pair whose first component is equal to this x. However, this obvious impression is incorrect. The expression above is equivalent to $\lambda x.\{(x', y) \mid (x', y) \in R\}$ with x' a fresh variable. That is, it takes in an x and reproduces an exact copy of the relation R.

Variable-slashing reduces this kind of mistake because it makes the above expression illegal. To see this, let me rewrite the expression in CPL without inserting the proper slashes: $\backslash x$ `=> {(`x, y`) | (`x, y`) <-` $R\}$. Now this expression is no longer closed because y has become a free variable. The CPL program that implements the obvious but incorrect meaning of the original expression is: $\backslash x$ `=> {(`x, y`) | (`$x, \backslash y$`) <-` $R\}$. The CPL program that implements the correct but obscured meaning of the original expression is: $\backslash x$ `=> {(`x, y`) | (`$\backslash x, \backslash y$`) <-` $R\}$. The absence and presence of the slash in front of the third x makes the difference very clear.

Examples: Convenience of pattern matching

It is generally agreed that pattern matching makes queries more readable. Here are some examples to illustrate CPL's pattern-matching mechanism.

Example. To illustrate partial-record patterns, here is a CPL query for finding the names of children who are ten years old:

```
primitive ten_year_olds ==
  \people => { x | (#name: \x, #age: 10, ...) <- people} ;
Result : Primitive ten_year_olds installed.
Type   : {(#name : ''1, #age: int; ''2)}->{''1}

ten_year_olds @ {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")};
Result : {"tom"}
Type   : {string}
```

Example. To illustrate layered patterns, here is the CPL query that returns those children who are ten years old (that is, not just their names):

```
primitive ten_year_olds' ==
  \people => { y | \y&(#name: \x, #age: 10, ...) <- people} ;
Result : Primitive ten_year_olds' installed.
Type   : {(#name:''1, #age:int; ''2)}->{(#name:''1, #age:int;''2)}

ten_year_olds' @ {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")};
Result : {(#name : "tom", #age : 10, #sex : "male")}
Type   : {(#name : string, #age : int, #sex : string)}
```

Example. To illustrate variable-as-constant patterns, consider generalizing `ten_year_olds`

to find names of children who are x years old, where x is to be given. Here is the query in CPL:

```
primitive x_year_olds ==
  (\people, \x) => { y | (#name: \y, #age: x, ...) <- people } ;
Result : Primitive x_year_olds installed.
Type   : (#1: {(#name : ''1, #age: ''2; ''3)}, #2: ''2)->{''1}

x_year_olds @ ( {(#name:"tom", #age:10, #sex:"male"), (#name:"liz",
#age: 5, #sex:"female"), (#name:"jim", #age:12, #sex:"male")}, 12) ;
Result : {"jim"}
Type   : {string}
```

Notice that the `10` in the `ten_year_olds` query is simply replaced by `x`, the input to be given. Since this occurrence of x is not slashed, it is not the introduction of a new variable. Rather it stands for the value that is supplied to the function as its second argument (that is, the `\x` argument). This kind of pattern is not found in any other pattern-matching language with which I am acquainted. (Prolog [176] does support a pattern mechanism based on unification which can be used to simulate my variable-as-constant patterns. However, Prolog is not a pattern-matching language. The task of matching Q against a pattern P is in finding a substitution θ so that $Q = (P)\theta$. The task of unifying Q and P is in finding a substitution θ so that $(Q)\theta = (P)\theta$. The two are clearly different.)

Without the variable-as-constant pattern mechanism, the same function would have to be written using an explicit equality test, producing a query that is quite different from `ten_year_olds`:

```
primitive x_year_olds' ==
  (\people,\x) => {y | (#name:\y, #age:\z, ...) <- people, z = x};
Result : Primitive x_year_olds' installed.
Type   : (#1: {(#name : ''1, #age: ''2; ''3)}, #2: ''2)->{''1}
```

Example. As the final pattern-matching example, here is a CPL query that computes the average salary of employees in departments. (This example uses an external primitive `average`. See Section 5.4 and Chapter 9 for a description of how to add a new external primitive to CPL.)

```
primitive ave_sal_by_dept == \DB =>
  {(#dept: x,
    #ave_sal: average @ {| y | (#dept:x, #sal:\y, ...) <- DB |})
  | (#dept: \x, ...) <- DB};
Result : Primitive ave_sal_by_dept installed.
Type : {(#sal:int, #dept:''10; ''3)}->{(#ave_sal:real, #dept:''10)}



ave_sal_by_dept @ {
  (#dept:  "cis", #emp: "john", #sal: 1000),
  (#dept:  "cis", #emp: "jeff", #sal: 1000),
  (#dept:  "cis", #emp: "jack", #sal:  400),
  (#dept: "math", #emp: "jane", #sal:  900),
  (#dept: "math", #emp: "jill", #sal:  600),
  (#dept:  "phy", #emp: "jean", #sal: 2000)};
Result : {(#stat: 2000.0, #dept: "phy"),
          (#stat: 750.0, #dept: "math"),
          (#stat: 800.0, #dept: "cis")}
Type   : {(#stat:real, #dept:string)}
```

Note the conversion to bag in the query, which captures the semantics of *group-by* in SQL. Without this conversion, then John and Jeff in the example input will cause the average salary in the CIS Department to be miscounted.

5.4 Other features of CPL

I have mentioned a few other features of CPL earlier on: it has a type inference system, it is extensible, and it has an optimizer. Extensibility and optimization are discussed in greater detail in later chapters. I use some simple examples to illustrate them here.

Types are automatically inferred

The type system is simpler than that of Ohori [154], Remy [165], and Jategaonkar and Mitchell [110]. The reason for this is that CPL does not have a record concatenation mechanism. For example, CPL infers that `ten_year_olds` has unique most general type `{(#name : ''1, #age: int; ''3)}->{''1}`.

Easy to add new primitives

The core of CPL is not a very expressive language. In fact, when restricted to sets, it is equivalent to the well-known nested relational algebra of Thomas and Fischer [183]. Therefore, it has to be augmented with extra primitives that reflect the needs of the applications that non-expert users are trying to solve. The extra primitives are provided by expert users who build them in the host language. Non-expert users only need to import them into CPL. This philosophy is demonstrated later, in Chapter 9.

It is very easy to extend CPL with new primitives. I illustrate this feature by showing how to insert a factorial function into CPL. The expert user first programs the factorial function `hostFact` in his host language, and registers it with the Kleisli query system as `factorial` by a simple library call in his host language. The host language is ML [144], it is not to be confused with CPL. (In ML, `F o G` stands for composition of functions `F` and `G` and `M.F` stands for the function `F` in the module `M`.)

```
DataDict.RegisterCompObj(
```

```
"factorial",
CompObjFunction.Mk(CompObjInteger.Mk o hostFact o CompObjInteger.Km),
TypeInput.ReadFromString "int -> int");
```

`DataDict.RegisterCompObj` is the registration routine. `TypeInput.ReadFromString` is the routine for converting a type specification given in a string to the internal format used by Kleisli. `CompObjFunction.Mk`, `CompObjInteger.Mk`, and `CompObjInteger.Km` are routines for converting between the Kleisli's and the host language's representations of complex objects. These routines are provided in the libraries of the Kleisli query system.

The non-expert user can then begin using the new primitive `factorial`.

```
factorial @ 5 ;
Result : 120
Type   : int
```

Easy to add new writers

To be useful a query language must be able to produce external data. CPL uses "writers" for writing external data in various format. It is easy to add new writers to CPL. There are five things associated with a writer. First is a function for connecting a text stream to the external data sink. Second is a de-tokenizer for converting Kleisli's token stream to a text stream in the required external format. Third is a string for identifying the writer. Fourth is a schema generator for generating the schema of the external data. Fifth is an input parameter type specification that describes how the external data sink is specified.

Below is an example program which adds the standard writer `StdOut` to CPL. The ML function `FileManager.WriterTab.Register` is the registration routine. The ML function `Tokenizer.TokenStreamToOutStream` is de-tokenizer for converting token stream to a text stream in Kleisli's standard exchange format.

```
FileManager.WriterTab.Register(
```

```
fn X => let val X = CompObjString.Km X
        in (X, open_out(X ^ ".val")) end,
fn (_,OS,TS) => Tokenizer.TokenStreamToOutStream(
                  TS,OS, fn _ => output(OS, "\n")),
"StdOut",
fn (X, T) => let
  val X = open_out((CompObjString.Km X) ^ ".typ")
  val T = Type.Stringify T
  val _ = output(X, T)
  val _ = output(X, "\n")
  val _ = close_out X
  in () end,
Type.String)
```

A non-expert user can use the `writefile` DATA `to` SINK `using` WRITER command for producing external data, as in the CPL example below.

```
writefile {1,2,3} to "temp" using StdOut;
Result: File temp written.
Type: {int}
```

Easy to add new scanners

To be useful a query language must be able to read external data. CPL uses "scanners" for reading external data in various format. It is easy to add new scanners to CPL. There are five things associated with a scanner. First is a generator function which returns the external data as a text stream in a chosen information exchange format. Second is a tokenizer for converting the text stream into token stream. Third is a string for identifying the scanner. Fourth is an input parameter type specification which describes how the external data source is specified. Fifth is schema reader for reading the schema of the external data.

Here is a program that adds the standard scanner `StdIn` to CPL. The ML function `FileManager.ScannerTab.Register` is the registration routine. The ML function `Tokenizer.InStreamToTokenStream` is the tokenizer for text stream in Kleisli's standard exchange format.

```
FileManager.ScannerTab.Register(
     fn X => let val X = Kleisli.CompObjString.Km X
             in (X, open_in (X ^ ".val")) end,
     Tokenizer.InStreamToTokenStream,
     "StdIn",
     Type.String,
     fn X => TypeInput.ReadFromFile(
             (Kleisli.CompObjString.Km X) ^ ".typ"))
```

After that, a non-expert user can read external data files in Kleisli's standard exchange format using the `readfile` NAME `from` SOURCE `using` SCANNER command. For example, the file `temp` written out earlier can now be read in.

```
readfile pmet from "temp" using StdIn;
Result: File pmet registered.
Type: {int}

pmet;
Result: {1, 2, 3}
Type: {int}
```

An extensible optimizer is available

CPL is equipped with an extensible optimizer. The optimizer does pipelining, joins, caching, and many other kinds of optimization. I illustrate it here on a very simple query. First, let me create a text file.

```
writefile {{(1,2), (3,4)}, {(5, 6), (7, 8)}} to "tmp" using StdOut;
Result : File tmp written.
Type   : {{(#1:int, #2:int)}}
```

Now we query the file by doing a flatten and a projection operation on it:

```
readfile db from "tmp" using StdIn;
Result : File db registered.
Type   : {{(#2:int, #1:int)}}

{ x | \X <- db, (\x, _) <- X } ;
Result : {1, 3, 5, 7}
Type   : {int}
```

Without the optimizer, the peak space requirement is memory to hold 4 integers and nothing gets printed until the entire set $\{1,3,5,7\}$ has been constructed. With the optimizer, the peak space requirement for this query is space for 1 integer and the first element of the output is printed instantly (while the rest of the output is still being computed). The reason is that the optimizer is sophisticated enough to push the projection on `(\x, _)` and the printing of `x` directly into the scanning of the input file `"tmp"`. (Note that the `readfile db from "tmp"` part of the query does not actually read the file; it merely establishes the file `"tmp"` as an input stream.)

A more detailed account of these extensibility features can be found in Chapter 9.

Part III

An engineer's drudgeries

Chapter 6

'Monadic' Optimizations

The evaluation of a query in a practical database has three phases. The first phase reads external data into memory and converts it into the right format for manipulation. The second phase performs the actual manipulation to satisfy the objective of the query. The final phase prints the result of the query. There are also three aspects in the cost of a query. The first is the amount of time it takes to complete the query. The second is the amount of memory space (disk space is ignored) it takes to evaluate the query. The third is the amount of time it takes before any portion of the result can be output; that is, response time.

Many existing treatments of query optimization (such as Fegaras [63]; Sheard and Fegaras [172]; Ullman [190]; and Trinder and Wadler [186]) did not explicitly consider reading of external data and printing of results. Translations between structured strings and databases were studied by Abiteboul, Cluet, and Milo [3]. They gave examples of some possible optimizations. It is not yet clear to me whether their examples are due to pre-determined circumstances or are instances of a more general technique. Freytag [68] is a more outstanding exception in that he explicitly considered transformation of scanning routines and their interaction with the more usual query operators. He used very sophisticated program transformation techniques for this purpose and he only considered flat relations. All of the

work above also did not explicitly discuss the three aspects, especially response time, of the cost of query evaluation.

This chapter investigates techniques for improving queries over nested collections, taking all three phases of evaluation and all three aspects of cost into account.

Organization

Section 6.1. There are two important methods for optimizing loops, both involve combining two loops into one (see Freytag [68] and Goldberg and Paige [72] for example). The first, called vertical loop fusion, is the fusion of two loops where the first loop produces the data consumed by the second loop. The second, called horizontal loop fusion, is the fusion of two independent loops iterating over the same collection. Another method for reducing the cost of loops is the migration of filters closer to generators (see Watt and Trinder [199] and Ullman [190] for examples). The performance of loops can also be improved by moving invariant code out of a loop (see Aho, Sethi, and Ullman [6] for example). This section suggests structural rewrite rules of sufficient generality to capture these methods.

Section 6.2. The input phase is captured abstractly as a process of converting an input stream into a complex object. Some scanning constructs for converting input tokens into complex objects are described. I present rewrite rules for improving queries in query languages enriched with these constructs. I discuss how excessive consumption of space caused by loading entire external files can be avoided using these rules.

Section 6.3. The output phase is captured abstractly as a process of converting a complex object into an output stream. Some printing constructs for converting complex objects to output tokens are described. I present rewrite rules for improving queries in query languages enriched with these constructs. When the result of a query is a large relation, it is desirable that the first few rows be printed as soon as possible while the remaining rows are still being computed. This property is achieved by giving the printing constructs a lazy operational semantics. The rewrite rules suggested in this section progressively pushes these

lazy printing operators into the query, giving rise to interesting interactions between the lazy and the eager mechanisms. In particular, execution is eager by default and laziness is introduced by rewrite rules. This strategy is in contrast to the tradition of lazy languages, where execution is lazy by default and eagerness is introduced by performing strictness analysis [97].

Section 6.4. The two previous sections deal with the situation of scanning an external stream followed by complex object manipulations and with the situation of complex object manipulations followed by printing. This section provides additional rewrite rules and programming constructs to take care of the situation where scanning is followed by printing. I also consider the situation where input token streams and output token streams are identified. This consideration leads to more rules to handle the fourth situation where printing is followed by scanning; however, no new programming construct is needed.

6.1 Structural optimizations

Basic optimizations

Several rewrite rules have already been given in Chapter 3. In this section, I discuss their effect as query optimization rules for $\mathcal{NRC}$. Let me list these rules and a few additional ones below. These rules are obtained by orienting equational axioms of $\mathcal{NRC}$ in a manner that reduces the amount of intermediate data.

- $(\lambda x.e_1)(e_2) \leadsto e_1[e_2/x]$
- $\lambda x.(e\ x) \leadsto e$ if x not free in e.
- $e \leadsto ()$ if $e : \mathit{unit}$ and e is not $()$.
- $\pi_i(e_1, e_2) \leadsto e_i$
- $(\pi_1\ e, \pi_2\ e) \leadsto e$

- $\mathit{if}\ \mathit{true}\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \rightsquigarrow e_1$
- $\mathit{if}\ \mathit{false}\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \rightsquigarrow e_2$
- $\mathit{if}\ (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3)\ \mathit{then}\ e_4\ \mathit{else}\ e_5 \rightsquigarrow$
  $\mathit{if}\ e_1\ \mathit{then}\ (\mathit{if}\ e_2\ \mathit{then}\ e_4\ \mathit{else}\ e_5)\ \mathit{else}\ (\mathit{if}\ e_3\ \mathit{then}\ e_4\ \mathit{else}\ e_5)$
- $e\ (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ e\ e_2\ \mathit{else}\ e\ e_3$
- $\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_2 \rightsquigarrow e_2$
- $(e,\ \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ (e, e_2)\ \mathit{else}\ (e, e_3)$
- $(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3,\ e) \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ (e_2, e)\ \mathit{else}\ (e_3, e)$
- $e \cup (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ e \cup e_2\ \mathit{else}\ e \cup e_3$
- $(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \cup e \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ e_2 \cup e\ \mathit{else}\ e_3 \cup e$
- $\{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \{e_2\}\ \mathit{else}\ \{e_3\}$
- $\bigcup\{e \mid x \in \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \bigcup\{e \mid x \in e_2\}\ \mathit{else}\ \bigcup\{e \mid x \in e_3\}$
- $\bigcup\{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid x \in e\} \rightsquigarrow \mathit{if}\ e_1\ \mathit{then}\ \bigcup\{e_2 \mid x \in e\}\ \mathit{else}\ \bigcup\{e_3 \mid x \in e\}$, if x not free in e_1.
- $\bigcup\{\{\} \mid x \in e\} \rightsquigarrow \{\}$
- $\bigcup\{e \mid x \in \{\}\} \rightsquigarrow \{\}$
- $\{\} \cup e \rightsquigarrow e$
- $e \cup \{\} \rightsquigarrow e$
- $e \cup e \rightsquigarrow e$
- $\bigcup\{\{x\} \mid x \in e\} \rightsquigarrow e$
- $\bigcup\{e_1 \mid x \in \{e_2\}\} \rightsquigarrow e_1[e_2/x]$
- $\bigcup\{e \mid x \in e_1 \cup e_2\} \rightsquigarrow \bigcup\{e \mid x \in e_1\}\ \cup\ \bigcup\{e \mid x \in e_2\}$

- $\bigcup\{e_1 \mid x \in e\} \cup \bigcup\{e_2 \mid x \in e\} \rightsquigarrow \bigcup\{e_1 \cup e_2 \mid x \in e\}$
- $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \rightsquigarrow \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$

The correctness of these rules can be easily ascertained.

Proposition 6.1.1 *If $e_1 \rightsquigarrow e_2$, then $e_1 = e_2$.* □

Although the rewrite system induced by the above rewrite rules is quite big, it has a desirable property: any sequence of applications of these rules leads to a normal form in a finite number of steps. Therefore an optimizer constructed using this system of rewriting is guaranteed to terminate, regardless of how it chooses to apply these rules. This strongly normalizing property allows the optimizer to concentrate on picking the most profitable sequence of rewriting, without worrying about getting into a loop.

Proposition 6.1.2 *The rewrite system induced by the above rules is strongly normalizing.* □

Observe that the proof on the conservative extension property of $\mathcal{NRC}$ in Section 3.1 uses a subset of the above rules. In the remainder of this section I present arguments showing that these rules are effective optimization rules. As a consequence, the normalization process used in Section 3.1 is also conservative over efficiency!

Loop fusion

The most costly construct in $\mathcal{NRC}$ is its loop construct $\bigcup\{e_1 \mid x \in e_2\}$. Assume that this construct is evaluated as follows. First evaluate e_2 into a set $\{o_1, \ldots, o_n\}$. Then evaluate each $e_1[o_i/x]$ into o_i'. Then form $o_1' \cup \ldots \cup o_n'$. Assume the cost of evaluating e_2 to be $\#(e_2)$. Assume, for simplicity, the cost of evaluating $e_1[o_i/x]$ to be $\#(e_1)$. Assume that the union of two sets takes constant effort 1 (this assumption is reasonable by supposing that duplicates

are not removed). The cost of evaluating the loop is, however, not $\#(e_2) + n \cdot \#(e_1) + n - 1$. There is an overhead of taking apart the set $\{o_1, \ldots, o_n\}$ that must also be accounted for. Assume the cost for traversing a set to be equal the its cardinality minus 1 (this assumption is reasonable since a good implementation should use no more than n links to connect up $n + 1$ items). Consequently the cost of the loop is $\#(e_2) + n \cdot \#(e_1) + n - 1 + n - 1$. (This cost function is admittedly rather naive. Nevertheless, it is indicative of the relative costs of different expressions.)

There are two well-known methods for optimizing loops, both involving combining two loops into one [72]. The first one is applicable when the first loop is a producer and the second loop is a consumer. Instead of building a separate set to keep the objects produced by the first loop and then pass this set to the second loop, the objects are pipelined directly to the second loop. This optimization is called vertical loop fusion. The second one is applicable when there are two independent loops over the same set. Instead of doing the first loop and then the second loop in a process requiring the set to be traversed twice, both loops are performed simultaneously. This optimization is called horizontal loop fusion.

The point of loop fusion is to reduce the amount of intermediate data, not unlike traditional database query optimization techniques that concentrate on reducing the number of columns and rows involved [190, 139]. For $\mathcal{NRC}$ the rule $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \rightsquigarrow \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$ is the only way to achieve vertical loop fusion, while $\bigcup\{e_1 \mid x \in e\} \cup \bigcup\{e_2 \mid x \in e\} \rightsquigarrow \bigcup\{e_1 \cup e_2 \mid x \in e\}$ appears to be the principal way to achieve horizontal loop fusion. (There are other more involved ways of doing horizontal loop fusion, but it is not very rewarding to describe them.) Their effectiveness with respect to the cost measure explained earlier is verified below.

Observation 6.1.3 *The cost of evaluating* $\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\}$ *exceeds the cost of evaluating* $\bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$ *by an amount roughly equal to twice the number of elements in the result of evaluating* e_3.

Explanation. Let e_3 evaluate to a set having a elements. For simplicity assume that each $e_2[o/y]$, where $o \in e_3$, evaluates to a set having b elements. Then the cost of the first

expression is $(\#(e_3)+a\cdot\#(e_2)+a-1+a-1)+((a\cdot b)\cdot\#(e_1)+(a\cdot b-1)+(a\cdot b-1))$. However, the cost of the second expression is $\#(e_3)+a\cdot(\#(e_2)+b\cdot\#(e_1)+b-1+b-1)+a-1+a-1$. The difference is $2\cdot a$. □

The improvement above comes from avoiding the overhead of having to construct the set $\bigcup\{e_2 \mid y \in e_3\}$ and from avoiding the overhead of having to dismantle that very same set immediately. This saving directly reduces the time taken to complete the query. If we assume that each object in $e_2[o/y]$ where $o \in e_3$ takes up a large amount of space, this saving also reduces the space consumption of the query. Response time is also improved indirectly.

Observation 6.1.4 *The cost of evaluating* $\bigcup\{e_1 \mid x \in e\} \cup \bigcup\{e_2 \mid x \in e\}$ *exceeds the cost of* $\bigcup\{e_1 \cup e_2 \mid x \in e\}$ *by an amount roughly equal to the sum of the cost of* e *and the number of elements in* e.

Explanation. Let e evaluate to a set having n elements. Then the cost of the first expression is $(\#(e)+n\cdot\#(e_1)+n-1+n-1)+(\#(e)+n\cdot\#(e_2)+n-1+n-1)+1$. However, the cost of the second expression is $\#(e)+n\cdot(\#(e_1)+\#(e_2)+1)+n-1+n-1$. The difference is $\#(e)+n+1$. □

Under a smarter system the cost of executing e twice in the observation above can be avoided. However, the need for traversing the set twice cannot be avoided. The horizontal fusion rule thus reduces the time taken to complete a query, although its impact is not as signifcant as the vertical fusion rule.

Examples

The system above of rewrite rules also generalizes many optimizations known for relational algebras. Following Trinder [184], I illustrate some of these improvements by examples. Let me use $\bigcup\{e_1 \mid x \in e_2 \textit{ where } e_3\}$ as a shorthand for $\bigcup\{\textit{if } e_3 \textit{ then } e_1 \textit{ else } \{\} \mid x \in e_2\}$. Since

$\bigcup\{e_1 \mid x \in e_2\}$ is the only loop construct in $\mathcal{NRC}$, the efficiency gained can be roughly estimated by comparing the number of loops before and after optimization.

- Combining a chain of projections. The query $\bigcup\{\{\pi_1 x\} \mid x \in \bigcup\{\{(\pi_1(\pi_2\ y),\ \pi_1\ y)\} \mid y \in R\}\}$ contains two consecutive projections. It rewrites to $\bigcup\{\{\pi_1(\pi_2\ y)\} \mid y \in R\}$. The optimized query is expected to execute faster than the original one because it has fused two projection loops into one.

- Combining a chain of selections. The query $\bigcup\{\{y\} \mid y \in \bigcup\{\{x\} \mid x \in R\ \mathit{where}\ p\ x\ \}\ \mathit{where}\ q\ y\ \}$ has two selection conditions that are applied one after another. It is rewritten to a conjunctive query $\bigcup\{\mathit{if}\ p(x)\ \mathit{then}\ \mathit{if}\ q(x)\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\}\ \mathit{else}\ \{\} \mid x \in R\}$. If the predicate p has $f\%$ selectivity, the improved query is expected to apply the predicate q about $(100-f)\%$ less often than the original version. Also, the two selection loops has been fused into one.

- Combining selection and projection. The query $\bigcup\{\{\pi_2\ x\} \mid x \in \bigcup\{\{y\} \mid y \in \bigcup\{\{(\pi_1(\pi_2\ z),\ \pi_1\ z)\} \mid z \in R\ \}\ \mathit{where}\ p(y)\ \}\ \}$ contains a selection sandwiched between two projections. It rewrites to $\bigcup\{\{\pi_1\ z\} \mid z \in R\ \mathit{where}\ p(\pi_1(\pi_2\ z),\ \pi_1\ z)\}$. The optimized query is likely to execute faster than the original query, because the optimized query has only one loop while the original query has three.

- Moving filter toward generator. The query $\bigcup\{\bigcup\{\mathit{if}\ p(x)\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid y \in S\} \mid x \in R\}$ contains a filter $p(x)$ that is far away from the generator $x \in R$. It rewrites to $\bigcup\{\mathit{if}\ p(x)\ \mathit{then}\ \bigcup\{e_1 \mid y \in S\}\ \mathit{else}\ \bigcup\{e_2 \mid y \in S\} \mid x \in R\}$. The filter has been moved immediately next to its generator in the optimized query. Suppose the selectivity of the filter is $f\%$, S has s elements, and R has r elements. The cost of the original query is roughly $r + s \cdot r$. The cost of the optimized query is only $r + s \cdot r \cdot f\%$.

- Subquery-to-join conversion. The query $\bigcup\{\bigcup\{\bigcup\{\{a\} \mid y \in \bigcup\{\{q_B\ b\} \mid b \in B\ \mathit{where}\ p_B\ b\}\} \mid z \in \bigcup\{\{q_C\ c\} \mid c \in C\ \mathit{where}\ p_C\ c\}\} \mid a \in A\ \mathit{where}\ p(a,y,z)\}$ contains two subqueries. It goes to $\bigcup\{\bigcup\{\bigcup\{\{a\} \mid c \in C\ \mathit{where}\ p_C\ c\} \mid b \in B\ \mathit{where}\ p_B\ b\} \mid a \in A\ \mathit{where}\ p(a,\ q_B\ b,\ q_C\ c)\}$. The two subqueries in the original query are converted into a join. Joins are preferable to subqueries because a lot of work has been done on

join optimization [163]. In any case, the original query contains five loops while the optimized query has only three.

Miscellaneous rules

Some of the rewrite rules above can cause certain expressions to be evaluated several times. To compensate for their effect, common subexpressions must be identified. There is currently no way to identify common subexpressions in $\mathcal{NRC}$. So one can contemplate introducing the new construct in Figure 6.1 and interpret $let\ x = e_1\, in\ e_2$ as $e_2[e_1/x]$. The

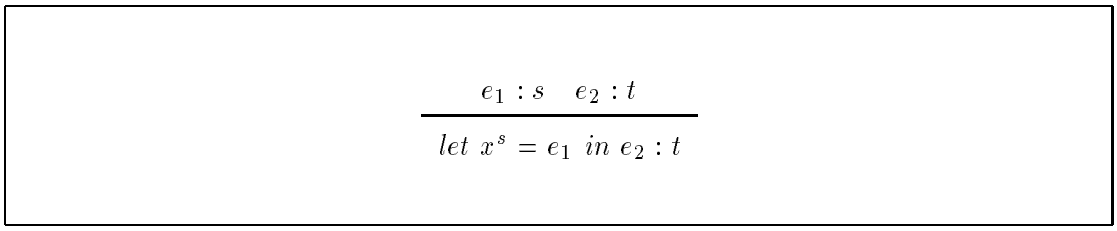


Figure 6.1: The *let*-construct.

obvious evaluation rule for $let\ x = e_1\ in\ e_2$ is first evaluate e_1 to an output N, store it in x and then evaluate e_2. The $let\ x = e_1\ in\ e_2$ construct is then used to take care of common subexpressions. The obvious rule has the form

$$(\cdots e \cdots e \cdots) \rightsquigarrow let\ z = e\ in\ (\cdots z \cdots z \cdots)$$

where the free variables of e form a subset of the free variables of $(\cdots e \cdots e \cdots)$ and e is not a variable or constant.

This construct gives us a third way to optimize a loop: code motion [6]. The idea is to migrate a block of invariant code out of a loop. It can be achieved by a rule of the form

$$\bigcup\{(\cdots e' \cdots) \mid x \in e\} \rightsquigarrow let\ y = e'\ in\ \bigcup\{(\cdots y \cdots) \mid x \in e\}$$

where the free variables of e' form a subset of the free variables of $\bigcup\{(\cdots e' \cdots) \mid x \in e\}$ and e' is not a variable or a constant. (Note that it is *incorrect* to use the simpler condition

that x is not free in e'.) It is easy to see that code motion yields a saving proportional to the number of elements in set e, unless the e's can be optimized better in place.

Discussion

In the work of Beeri and Kornatzky [19], Trinder [184, 185], and Osborn [156], they suggested a number of general identities that can be used for query optimization. Their identities must be used carefully because not all sequences of rewriting using them are guaranteed to reach a normal form. That is, the optimizers must incorporate some mechanism for avoiding non-termination. The rules of Beeri and Kornatzky are for a language that is more general than the calculus of this report. Their identities are very powerful. However, the generality of their identities may cause difficulty in the automation of their rules. This difficulty is precisely the reason that many implementations of program transformation systems, such as Darlington [51] and Firth [67], require human guidance.

In a series of papers [194, 195, 197], Wadler proposed general loop fusion techniques and proved their effectiveness in general functional programming systems. Freytag [68] demonstrated their effectiveness in the specific context of flat relational databases. In both cases, they worked with a powerful recursive programming language. It should be mentioned that Freytag's rewrite system has the Church-Rosser property. Hence all rewritings lead to a unique normal form. However, Freytag has to sacrifice certain optimizations to achieve this property. My system does not enjoy this property due to the presence of rules such as $(e,\ \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \leadsto \mathit{if}\ e_1\ \mathit{then}\ (e, e_2)\ \mathit{else}\ (e, e_3)$. Having more rules gives me some freedom to incorporate a cost model for picking which rules to apply during rewriting.

I should mention the work of Fegaras [63] and Sheard and Fegaras [172]. They identified a special form of structural recursion that is slightly more general than my homomorphic restriction and developed very general rules for performing vertical loop fusion for it. There are very strong parallels between our approaches. However, they emphasized types that are sums-of-products while I concentrate on collection types such as sets and bags.

Erwig and Lipeck [60] provided a set of rules somewhat similar to that of Beeri and Kornatzky's. In addition, they suggested a strategy for using their rules. The strategy is similar the rule of thumb described in database texts like Ullman [190] and Maier [139]. The strategy is very simple and there is a possibility that, after carrying out rewriting according to it, some of the rules might still be applicable. Hence it might not realize fully the improvements that can be gained by a more clever application of their rules. In contrast, my procedure is very thorough; it always reduces a query to a normal form. Normal forms are desirable because they are usually simpler than non-normal forms. It is thus easier to estimate their costs and to perform further processing on them.

6.2 Scan optimizations

Input token streams

External data must be read and converted into a complex object prior to being queried. The conventional input conversion process is usually a routine that reads the external data and simply produces the corresponding complex object. There are two principal shortcomings of using such an input conversion process. First, a potentially large amount of space must be allocated for storage of the complex object in spite of the likelihood that the complex object will soon be dismantled for subsequent processing. Second, the input conversion process is a black box and prevents profitable migration of some of the subsequent operations on the complex object into it. This section investigates the opening up of the input conversion process and its effect on query optimization.

External data is regarded as a list of tokens. Tokens are essentially objects of base types and punctuation symbols: `(`, `)`, `,`, {, and }. A complex object is represented as external data in the obvious fashion. For example, the external data representing the object $\{(1,2), (3,4)\}$ is the following sequence of tokens: {, `(`, `1`, `,`, `2`, `)`, `,`, `(`, `3`, `,`, `4`, `)`, }. The space occupied by external data is disregarded. An input stream is an object representing a subfix of such a list, representing the remaining portion of the external data to be processed. (See Field

and Harrison [66] on the use of lazy data structures such as streams in programming. See Henderson [92] and Kelly [112] on the use of streams and process network in concurrent systems.)

An efficient implementation of input streams should have the following properties: (1) An input stream should occupy a small constant amount of space. (2) It should provide a constant time function `getInputToken` such that `getInputToken`(S) produces the first token on the input stream S. (3) It should provide a constant time function `skipInputToken` such that `skipInputToken`(S) produces an input stream S' obtained by skipping over the first token on S. (4) It should provide a constant time function `skipInputObj` such that `skipInputObj`(S) produces an input stream S' obtained by skipping over a prefix of S where the prefix represents a complex object. (5) It should be pure in the sense that it must not exhibit any observable side effects. (In contrast, the notion of streams in a language such as the Standard ML of New Jersey [11] is not pure.) It is not possible to achieve the above ideal, especially the fourth item. Nevertheless, it is possible to come quite close in practice.

It should be stressed that while an input stream represents an external datum, it does not have to contain the entire sequence of tokens at any one time. It merely has to produce the tokens in sequence when `getInputToken`, `skipInputToken`, or `skipInputObj` are applied to it. In other words, it lazily [66] brings in portions of the external datum.

Scanning constructs

The opening up of the input conversion process can be achieved by adding to $\mathcal{NRC}$ the new constructs listed in Figure 6.2, where $[\![s]\!]$ is the type for input streams representing complex objects of type s. While $[\![s]\!]$ is added to the type system, the collection of complex object types remain unchanged. A complex object type is still a type built entirely from sets, pairs, and base types; that is, it has neither $[\![\cdot]\!]$ nor arrows.

I present the semantics of the new constructs below. Note that these constructs manipulate input streams, as opposed to external data.

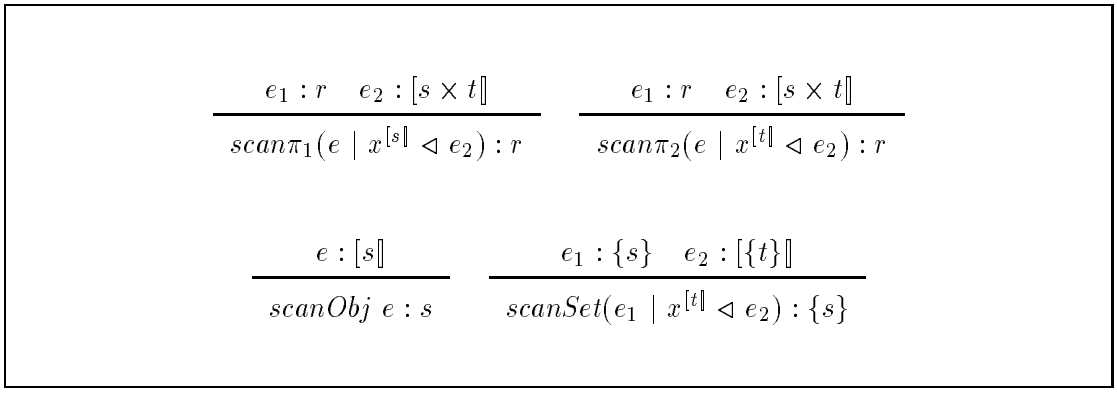


Figure 6.2: The constructs for input streams.

- The *scanObj* e construct requires e to be a input stream whose prefix represents an object o of complex object type s. The result of *scanObj* e is object o.

- The $scan\pi_1(e_1 \mid x \lhd e_2)$ construct requires e_2 to be an input stream whose prefix represents a pair (o_1, o_2). The result of the whole expression is $e_1[O/x]$ where O is the portion of the input stream representing o_1. Intuitively, O is obtained from e_2 by skipping over the initial left bracket (of the prefix (o_1, o_2) of the input stream e_2.

- The $scan\pi_2(e_1 \mid x \lhd e_2)$ construct requires e_2 to be an input stream whose prefix represents a pair (o_1, o_2). The result of the whole expression is $e_1[O/x]$ where O is the portion of the input stream representing o_2. Intuitively, O is obtained from e_2 by skipping over the initial fragment (o_1, of the prefix (o_1, o_2) of the input stream e_2.

- The construct $scanSet(e_1 \mid x \lhd e_2)$ requires e_2 to be an input stream whose prefix represents a set $\{o_1, \ldots, o_n\}$. The result of the whole expression is $f(O_1) \cup \ldots \cup f(O_n)$ where f is the function $\lambda x.e_1$ and O_i is the portion of the input stream representing o_i. Intuitively, each O_i is obtained from e_2 by skipping over the initial fragment $\{o_1, \ldots, o_{i-1},$ of the prefix $\{o_1, \ldots, o_n\}$ of the input stream e_2. The point of binding x to input streams instead of to objects is an important one. If x is required to bind to objects, then it is necessary to scan and hold the entire object in memory. However, by making x a stream, this complete loading is avoided.

The functions `getInputToken`, `skipInputToken`, and `skipInputObj` can be used to implement the constructs above. For illustration, I describe the operational behavior of $scanSet(e_1 \mid x \lhd e_2)$. First evaluate e_2 into an input stream S representing a set $\{o_1, \ldots, o_n\}$; assume this step has a cost $\#(e_2)$. Then use `skipInputToken` to skip over the opening { to get the stream S_1; assume this step has cost 1. Then evaluate $e_1[S_1/x]$ into a set O_1; assume this step has cost $\#(e_1)$. Then use `skipInputObj` on S_1 to skip over the object o_1; assume this step has cost 1. Then use `getInputToken` to see if the next token is the closing } or is the comma `,`; assume this step has cost 1. If it is a comma, use `skipInputToken` to skip over it to obtain the input stream S_2. Then evaluate $e_1[S_2/x]$ into a set O_2. Repeat the procedure to obtain sets O_3, ..., O_n until the matching closing } is encountered. Then form $O_1 \cup \ldots \cup O_n$; assume this step has cost $n - 1$. The cost of $scanSet(e_1 \mid x \lhd e_2)$ is easily seen to be $\#(e_2) + n \cdot \#(e_1) + n + n + n + n - 1$, where n is the cardinality of the set represented by the input stream e_2.

Scan optimization

The *scanObj e* construct is essentially the conventional scan-and-convert routine. However, the parameterization in the other constructs opens up the input conversion process. It is these other constructs that I exploit in my query optimizer obtained by appending the following rewrite rules to the system given in Section 6.1.

- $e(scan\pi_i(e_2 \mid x \lhd e_3)) \leadsto scan\pi_i(e\ e_2 \mid x \lhd e_3)$
- $\pi_i(scanObj\ e) \leadsto scan\pi_i(scanObj\ x \mid x \lhd e)$
- $\bigcup\{e_1 \mid x \in scanObj\ e_2\} \leadsto scanSet(e_1[(scanObj\ y)/x] \mid y \lhd e_2)$
- $\bigcup\{e_1 \mid x \in scanSet(e_2 \mid y \lhd e_3)\} \leadsto scanSet(\bigcup\{e_1 \mid x \in e_2\} \mid y \lhd e_3)$
- $scanSet(e_1 \mid x \lhd e) \cup scanSet(e_2 \mid x \lhd e) \leadsto scanSet(e_1 \cup e_2 \mid x \lhd e)$

These rules are sound.

Proposition 6.2.1 *If* $e_1 \rightsquigarrow e_2$, *then* $e_1 = e_2$. □

The last two rules are for vertical loop fusion and horizontal loop fusion. Their effectiveness are discussed in the two propositions below.

Observation 6.2.2 *The cost of evaluating* $\bigcup\{e_1 \mid x \in scanSet(e_2 \mid y \lhd e_3)\}$ *exceeds the cost of evaluating* $scanSet(\bigcup\{e_1 \mid x \in e_2\} \mid y \lhd e_3)$ *by an amount roughly twice the number of elements in the set represented by the input stream* e_3.

Explanation. Assume e_3 represents a set having a elements. Assume each $e_2[S_i/y]$ is a set having b elements, where S_i is a suffix of the stream e_3 after skipping $i-1$ objects. Then the cost of the first expression is $(\#(e_3)+a\cdot\#(e_2)+4\cdot a-1)+(a\cdot b)\cdot\#(e_1)+(a\cdot b)-1+(a\cdot b)-1$. However, the cost of the second expression is $\#(e_3)+a\cdot(\#(e_2)+b\cdot\#(e_1)+b-1+b-1)+4\cdot a-1$. The difference is roughly $2\cdot a$. □

Observation 6.2.3 *The cost of evaluating* $scanSet(e_1 \mid x \lhd e) \cup scanSet(e_2 \mid x \lhd e)$ *exceeds the cost of evaluating* $scanSet(e_1 \cup e_2 \mid x \lhd e)$ *by an amount approximately equal to the sum of the cost of evaluating* e *and thrice the number of elements in the set represented by* e.

Explanation. Assume e represents a set having n elements. The cost of the first expression is $(\#(e) + n \cdot \#(e_1) + 4 \cdot n - 1) + (\#(e) + n \cdot \#(e_2) + 4 \cdot n - 1)$. However, the cost of the second expression is $(\#(e) + n \cdot (1 + \#(e_1) + \#(e_2)) + 4 \cdot n - 1)$. The difference is roughly $\#(e) + 3 \cdot n$. □

The saving in vertical loop fusion comes from having avoided the need to explicitly assemble and dissemble the set $scanSet(e_2 \mid y \lhd e_3)$. This directly reduces the time for the query to complete and the space requirement. The saving in horizontal loop fusion comes from scanning the stream e only once. This reduces the time requirement. The two examples below provide more specific illustration of how space consumption is reduced by vertical loop fusion.

- Combining projection with scan. The query $\bigcup\{\{\pi_1\ x\} \mid x \in scanObj\ R\}$ first scans the input stream R to build a set of pairs and then returns the first components of these pairs. It is rewritten to $scanSet(\{scan\pi_1(scanObj\ y \mid y \lhd x)\} \mid x \lhd R)$. The improved query performs the projection while scanning the stream. As a result, assuming both components of the pairs occupy an equal amount of space, space consumption is reduced by 50%. Furthermore, the time required by the original query for first assembling the external data into a complex object is eliminated in the improved query.

- Combining selection with scan. The query $\bigcup\{\ if\ p(x)\ then\ \{x\}\ else\ \{\} \mid x \lhd scanObj\ R\}$ first scans the input stream R to build a set and then extracts those items that satisfy the predicate p. This query is rewritten to $scanSet(if\ p(scanObj\ y)\ then\ \{scanObj\ y\}\ else\ \{\} \mid y \lhd R)$. The improved query performs the selection while scanning the input stream. As a result, assuming the predicate has 50% selectivity, the amount of space consumed is reduced by 50%. Moreover, the time required by the original query for first assembling the external data into a complex object is eliminated in the improved query.

Discussion

Freytag [68] explicitly considered scanning routines during query optimization. Both his queries and scanning routines are expressed in a general functional language. His optimizer has the potential of expressing very efficient algorithms answering queries. However, this potential can only be realized by carrying out very sophisticated analysis on queries. In comparison, my optimizer uses much simpler analysis to produce appreciable improvement in queries.

Abiteboul, Cluet, and Milo [3] considered translation between structured strings and databases. They gave examples where queries are optimized by pushing some operations down to the scanning level. Their approach is more general than mine because they perform scanning based on description of external data that are not fixed beforehand. On the other

hand, their treatment of how to carry out the optimization is not very satisfactory. It is not unreasonable to envision a technique to tokenize their external data into an input stream of the form manipulable by my constructs. My optimizer can then be used to perform the optimization. Such an approach is more modular than theirs.

6.3 Print optimizations

Output token streams

The evaluation of a query is only useful when the result is written out. This requires a process of converting a complex object into external data. Such a process is usually a routine that takes in a complex object and then prints out some string-based representation of it. There are two shortcomings of using such a conversion process. First, a potentially large amount of space must be allocated for storage of the complex object in spite of the fact that it is immediately dismantled and written out. Second, nothing is written out until the whole complex object is materialized; this results in a long wait for the first output character to be written out (on the display screen). This section investigates the opening up of the output conversion process and its effect on query optimization.

Recall that external data is regarded as a list of tokens here. An output stream is an object representing a subfix of such a list, representing the portion of a complex object that is to be written out. A good implementation of output streams should have the following properties: (1) An output stream should occupy a small constant amount of space. (2) It should provide a function `getOutputToken` such that `getOutputToken`(S) returns the first token on the output stream S. (3) It should provide a function `skipOutputToken` such that `skipOutputToken`(S) returns an output stream S' obtained by the output stream S by skipping over the first token. (4) It should be pure in the sense that it exhibits no observable side effects. It is impossible to achieve all the properties above, especially the first item. Nevertheless, it is possible to come quite to close to it in practice.

It should be stressed that while an output stream represents a complex object, it does not have to contain the entire sequence of tokens representing that object. It merely has to be able to produce those tokens in sequence when `getOutputToken` and `skipOutputToken` are applied to it. Hence an output stream lazily [66] produces the portion of the complex object that needs to be written out.

Printing constructs

The opening up of the output conversion process is achieved by augmenting $\mathcal{NRC}$ with the constructs listed in Figure 6.3, where $[\![s]$ is the type for output stream representing complex objects of type s. While $[\![s]$ is added to the type system, the collection complex object types remain unchanged. A complex object type is still a type built entirely from sets, pairs, and base types; that is, it has neither $[\![\cdot]$ nor arrows.

$$\frac{}{putEmptySet^s : [\![\{s\}]} \qquad \frac{e : [\![s]}{putSingletonSet\ e : [\![\{s\}]} \qquad \frac{e_1 : [\![\{s\}] \quad e_2 : [\![\{s\}]}{putUnionSet(e_1, e_2) : [\![\{s\}]}$$

$$\frac{e : s}{putObj\ e : [\![s]} \qquad \frac{e_1 : [\![s] \quad e_2 : [\![t]}{putPair(e_1, e_2) : [\![s \times t]} \qquad \frac{e_1 : [\![\{s\}] \quad e_2 : \{t\}}{putSet(e_1 \mid x^t \in e_2) : [\![\{s\}]}$$

Figure 6.3: The constructs for output streams.

Note that these constructs manipulate output streams as oppose to printing out external data. Their semantics is given below.

- The *putObj* e construct produces an output stream representing the complex object e.
- The $putPair(e_1, e_2)$ construct produces an output stream whose first token is (, fol-

lowed by tokens on the output stream e_1, followed the token `,`, followed by tokens on the output stream e_2, followed by the token `)`.

- The *putEmptySet* construct produces an output stream consisting of the token { followed by the token }.

- The *putSingletonSet* e construct produces an output stream whose first token is {, followed by tokens on the output stream e, followed by the token }.

- The $putUnionSet(e_1, e_2)$ expects e_1 to be an output stream representing a sequence of tokens of the form {, o_1, `,`, ..., `,`, o_n, } and expects e_2 to be an output stream representing a sequence of tokens of the form {, o'_1, `,`, ..., `,`, o'_m, }. It produces an output stream representing the following sequence of tokens: {, o_1, `,`, ..., `,`, o_n, `,`, o'_1, `,`, ..., `,`, o'_m, }. That is, it strips the closing set-bracket } from e_1 and the opening set-bracket { from e_2 and then concatenating the two resulting streams, inserting a comma `,` if necessary. I should remark that it is not the duty of *putUnionSet* to eliminate duplicates.

- The $putSet(e_1 \mid x \in e_2)$ construct has the following semantics. Suppose e_2 is the set $\{o_1, \ldots, o_n\}$ and $\lambda x.e_1$ is the function f. Then it produces the output stream $putUnionSet(f(o_1), putUnionSet(\ \ldots, putUnionSet(f(o_{n-1}), f(o_n))\ldots))$.

The functions `getOutputToken` and `skipOutputToken` can be used to implement the constructs above. The operational semantics I have in mind for the above constructs is a mixture of lazy and eager evaluation. I avoid a detailed description here and provide a simplified description instead. First an expression $e : [\![s]$ is evaluated into an output stream P. Then P is passed to a print-loop that repeatedly executes the steps: (1) apply `getOutputToken` to the current output stream to get the current token; (2) display the token thus obtained; and (3) apply `skipOutputToken` to the current output stream to advance it by one token. Hence the behavior of the execution of $e : [\![s]$ can be considered in two stages: the evaluation of $e : [\![s]$ to P; and the execution of `getOutputToken`(P).

Mixed evaluation

To get a picture of the evaluation of $e : [\![s]$ to P, let me introduce the output format given by the following grammar:

$$\begin{aligned} P, Q ::= \quad & putObj\ M \mid putEmptySet \mid putSingletonSet\ M \\ \mid \quad & putUnionSet(P, Q) \mid putSet(e \mid x \in M) \mid putPair(P, Q) \end{aligned}$$

where $M, N ::= c \mid (M, N) \mid \{\} \mid \{M\} \mid M \cup N$. The expression $e : [\![s]$ is reduced using a call-by-value eager strategy [93] to an output format. From the output format it should be clear that $putSet(e_1 \mid x \in e_2)$ is a partly lazy construct: it evaluates e_2 completely and then suspends in the state $P \triangleq putSet(e_1 \mid x \in M)$, where M is a tree-like representation of the set $\{o_1, \ldots, o_n\}$. All other constructs are intended to be eager.

Now the print-loop is entered. In step (1), `getOutputToken`(P) is executed, which leads to P being split into $e_1[o_1/x]$ and $putSet(e_1 \mid x \in \{o_2, \ldots, o_n\})$. Then $e_1[o_1/x]$ is again evaluated into an output format P'. However, $putSet(e_1 \mid x \in \{o_2, \ldots, o_n\})$ is suspended. Then `getOutputToken` is applied to P' to extract its first token. In step (2), this token is printed. In step (3), the effect is equivalent to applying `skipOutputToken` to P' to get the output stream P''. The process now repeats with the new output stream equivalent to $putUnionSet(P'', putSet(e_1 \mid x \in \{o_2, \ldots, o_n\}))$. The net effect is that `getOutputToken` is next applied to P'' to extract the second token to be printed; this process is repeated until the tokens on $e_1[o_1/x]$ are exhausted; then $putSet(e_1 \mid x \in \{o_2, \ldots, o_n\})$ is accessed until all tokens are consumed.

Print optimization

The construct $putObj\ e$ is essentially the conventional convert-and-print routine. However, the parameterization in other constructs opens up the output conversion process. It is these other constructs that I exploit in my optimizer obtained by appending the following rewrite rules to the system given in Section 6.1.

- $putObj(e_1, e_2) \rightsquigarrow putPair(putObj\ e_1,\ putObj\ e_2)$
- $putObj\{\} \rightsquigarrow putEmptySet$
- $putObj\{e\} \rightsquigarrow putSingletonSet\ (putObj\ e)$
- $putObj(e_1 \cup e_2) \rightsquigarrow putUnionSet(putObj\ e_1, putObj\ e_2)$
- $putObj \bigcup\{e_1 \mid x \in e_2\} \rightsquigarrow putSet(putObj\ e_1 \mid x \in e_2)$
- $putSet(e \mid x \in \{\}) \rightsquigarrow putEmptySet$
- $putSet(e \mid x \in e_1 \cup e_2) \rightsquigarrow putUnionSet(putSet(e \mid x \in e_1),\ putSet(e \mid x \in e_2))$
- $putSet(e_1 \mid x \in \{e_2\}) \rightsquigarrow e_1[e_2/x]$
- $putSet(e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}) \rightsquigarrow putSet(putSet(e_1 \mid x \in e_2) \mid y \in e_3)$
- $putUnionSet(putSet(e_1 \mid x \in e),\ putSet(e_2 \mid x \in e)) \rightsquigarrow$
  $putSet(putUnionSet(e_1,\ e_2) \mid x \in e)$

The above rules are sound in the following sense:

Proposition 6.3.1 *Let two output streams be regarded as equivalent when they represent the same complex object. Then* $e_1 \rightsquigarrow e_2$ *implies* $e_1 = e_2$. □

The response time of a query is the time taken for the first token of the result to appear on the output stream (and get printed). A rough measure of the response time of $putSet(e_1 \mid x \in e_2)$ is $\#(e_2) + \#(e_1)$. Note that we are not interested in how fast the query completes, but how fast the first few characters appear on the screen. The effect of the rule corresponding to vertical loop fusion is demonstrated in the following proposition.

Observation 6.3.2 *The response time of* $putSet(e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\})$ *is slower than that of* $putSet(putSet(e_1 \mid x \in e_2) \mid y \in e_3)$ *by approximately the product of the number of elements in* e_3 *and the cost of evaluating* $e_2[o/x]$*, where* o *is a typical element in* e_3.

Explanation. Assume e_3 is a set having a elements. The response time of the first expression is estimated at $(\#(e_3) + a \cdot \#(e_2) + a - 1 + a - 1) + \#(e_1)$. The response time of the second expression is estimated at $\#(e_3) + \#(e_2) + \#(e_1)$. The difference is roughly $(a-1) \cdot (\#(e_2) + 2)$. □

The improvement in response time is significant. There is also a good reduction in space consumption because the set $\bigcup\{e_2 \mid y \in e_3\}$ is not constructed in the improved query. The overhead of suspending and re-activating subexpressions can affect the total time taken for the query to complete. This aspect of the performance of my mixed strategy tends to be better than a fully lazy strategy. However, it can be worse than a fully eager strategy if input data and intermediate data are small enough to fit entirely into memory.

6.4 Print-scan optimizations

Copying constructs

There is still an unsatisfactory aspect in the current set up. Consider *putObj*(*scanObj* e_2). There is currently no way in my language to avoid reading in the entire stream e_2 to assemble the object *scanObj* e_2 and then immediately dismantle it to print it out. This calls for some new constructs for combining the input conversion process and the output conversion process. To ease the fusion of input conversion and output conversion, I suggest the constructs listed in Figure 6.4.

The semantics of these new constructs is given below.

- The *putscanObj* e construct expects e to be an input stream whose prefix represents a complex object o of type s. It denotes the output stream representing the same complex object o.
- The *putscan*$\pi_1(e_1 \mid x \lhd e_2)$ construct expects e_2 to be an input stream whose prefix represents a pair (o_1, o_2). It denotes the output stream $e_1[O/x]$, where O is the portion

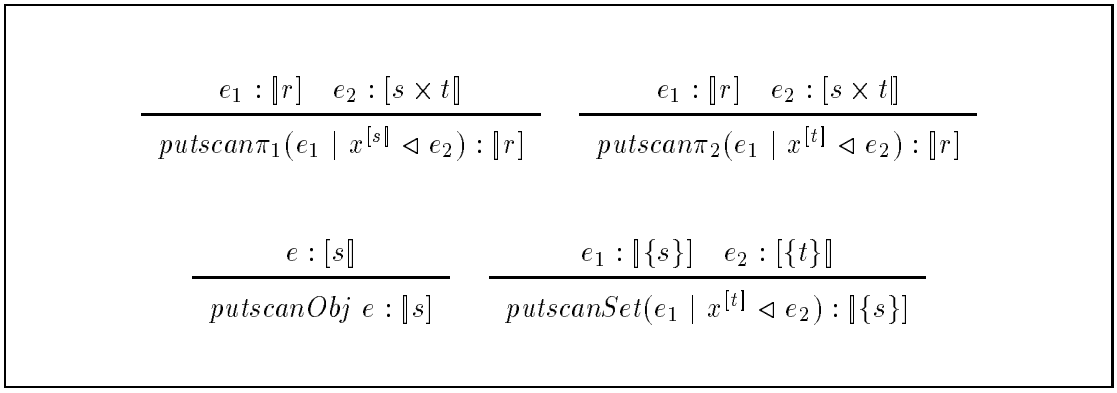


Figure 6.4: The constructs for stream interactions.

of the input stream representing o_1. Intuitively, O is obtained by skipping over the opening left-bracket of the input stream e_2.

- The $putscan\pi_2(e_1 \mid x \lhd e_2)$ construct expects e_2 to be an input stream whose prefix represents a pair (o_1, o_2). It denotes the output stream $e_2[O/x]$ where O is the portion of the input stream representing o_2. Intuitively, O is obtained from e_2 by skipping over the initial fragment $(o_1,$.

- The $putscanSet(e_1 \mid x \lhd e_2)$ construct requires e_2 to be an input stream whose prefix represents a set $\{o_2, \ldots, o_n\}$. The whole expression denotes the output stream $putUnionSet(f(O_1),\ \ldots,\ putUnionSet(f(O_{n-1},\ f(O_n))\ldots)$, where f is the function $\lambda x.e_1$ and O_i is the portion of the input stream representing o_i. Intuitively, each O_i is obtained from e_2 by skipping over the initial fragment $\{o_1, \ldots, o_{i-1},$ of the prefix $\{o_1, \ldots, o_n\}$.

Mixed evaluation

To give a simplified account of the operational behavior these constructs, I need to add a few more output formats:

$$P ::= \ \ldots \mid putscanObj\ y \mid putscan\pi_i(e_1 \mid x \lhd y) \mid putscanSet(e_1 \mid x \lhd y)$$

(In a real implementation the y above will be names of input streams or pointers to files. In this dissertation, regard them either as free variables or as constants standing for input streams.) The evaluation of an expression $e : [\![s]$ again has two stages. The first stage uses an eager call-by-value strategy to reduce $e : [\![s]$ to an output format $P : [\![s]$. Observe that the three new output formats introduced above are also partly lazy. The second stage is the print-loop described earlier. Let me describe the behavior of the print-loop on the output format $P \triangleq putscanSet(e \mid x \lhd R)$, where R is an input stream representing $\{o_1, \ldots, o_n\}$. In step (1), `getOutputToken`(P) is executed. This causes P to be split into $e[O/x]$ and $putscanSet(e \mid x \lhd R')$, where O is the portion of R representing o_1 and R' is the portion of R representing $\{o_2, \ldots, o_n\}$. Then $e[O/x]$ is evaluated to an output format P'. However $putscanSet(e \mid x \lhd R')$ is suspended. Then `getOutputToken` is applied to P' to extract its first token. In step (2), this token is printed. In step (3), the effect is equivalent to applying `skipOutputToken` to the output stream P' to get the output stream P''. The process now repeats with the new output stream equivalent to $putUnionSet(P'', putscanSet(e \mid x \lhd R'))$. The net effect is that `getOutputToken` is next applied to P'' to extract the second token to be printed; this process is repeated until the tokens on $e[O/x]$ are exhausted; then $putscanSet(e \mid x \lhd R')$ is accessed until all tokens are consumed.

Copy optimization

The *putscanObj e* construct is essentially a conventional file copy routine. The other new constructs are parameterized and provide some opportunity for optimization. The additional rules I have in mind are listed below.

- $putObj(scanObj\ e) \leadsto putscanObj\ e$
- $putObj(scan\pi_i(e_1 \mid x \lhd e_2)) \leadsto putscan\pi_i(putObj\ e_1 \mid x \lhd e_2)$
- $putObj(scanSet(e_1 \mid x \lhd e_2)) \leadsto putscanSet(putObj\ e_1 \mid x \lhd e_2)$
- $putSet(e_1 \mid x \in scanSet(e_2 \mid y \lhd e_3)) \leadsto putscanSet(putSet(e_1 \mid x \in e_2) \mid y \lhd e_3)$
- $putSet(e_1 \mid x \in scanObj\ e_2) \leadsto putscanSet(e_1[(scanObj\ y)/x] \mid y \lhd e_2)$

- $putUnionSet(putscanSet(e_1 \mid x \lhd e),\ putscanSet(e_2 \mid x \lhd e))$

  $\rightsquigarrow putscanSet(putUnionSet(e_1, e_2) \mid x \lhd e)$

The above rules are sound in the following sense:

Proposition 6.4.1 *Let two output streams be equivalent when they represent the same complex object. Then* $e_1 \rightsquigarrow e_2$ *implies* $e_1 = e_2$. □

For simplicity, I assume the response time of the *putscanSet* construct to be same as the *putSet* construct; hence the response time of $putscanSet(e_1 \mid x \lhd e_2)$ is roughly $\#(e_2) + \#(e_1)$. Similarly, I assume the total time of *putscanSet* and *putSet* to be same as the *scanSet* construct; hence the total time of $putscanSet(e_1 \mid x \lhd e_2)$ is roughly $\#(e_2) + n \cdot \#(e_1) + 4 \cdot n - 1$. Using these estimates the improvement achieved by some of the above rules can be calculated. I provide below the improvement from the vertical loop fusion rule, where it is clearly shown that the *putscanSet* construct effectively combines the response time improvement of *putSet* and the total time improvement of *scanSet*.

Observation 6.4.2 *The response time of* $putSet(e_1 \mid x \in scanSet(e_2 \mid y \lhd e_3))$ *is slower then that of* $putscanSet(putSet(e_1 \mid x \in e_2) \mid y \lhd e_3)$ *by approximately the product of the number of elements in* e_3 *and the time it takes to evaluate* e_2. *Moreover, the total time of the former is longer than the latter by approximately equal to the number of elements in* e_3.

Explanation. Assume e_3 is an input stream representing a set having a elements. Assume each $e_2[o/y]$, where o is an element of the set represented by e_3, yields a set having b elements. Then the response time of the first expression is $(\#(e_3) + a \cdot \#(e_2) + 4 \cdot a - 1) + \#(e_1)$. However, the response time of the second expression is $\#(e_3) + \#(e_2) + \#(e_1)$. The difference in response time is $(a-1) \cdot \#(e_2) + 4 \cdot a - 1$. Similarly, the total time of the first expression is $(\#(e_3) + a \cdot \#(e_2) + 4 \cdot a - 1) + (a \cdot b \cdot \#(e_1) + 4 \cdot a \cdot b - 1)$. However, the total time of the second expression is $(\#(e_3) + a \cdot (\#(e_2) + b \cdot \#(e_1) + 4 \cdot b - 1) + 4 \cdot a - 1)$. The difference in total time is $a - 1$. □

Monad of token stream

In the last three sections I distinguish between input token stream $[\![s]\!]$ and output token stream $\langle\!|s|\!\rangle$. This distinction has been useful for explaining the introduction of the various token stream constructs. However, it is reasonable to drop the distinction and to identify both of them as token stream $\langle\!|s|\!\rangle$. I gather in Figure 6.5 the token stream constructs presented in earlier sections. Since $[\![s]\!] = \langle\!|s|\!\rangle = \langle\!|s|\!\rangle$ now, the conversion construct $putscanObj^s$ is redundant. I omit it from the figure.

The token stream monad fragment of the table constitutes a complete physical language at an extremely low level. It corresponds strongly to the abstract language $\mathcal{NRC}$. In fact, the same kind of equational reasoning we have performed on $\mathcal{NRC}$ can be performed on this low level physical language — a feat that is quite remarkable. This physical language and $\mathcal{NRC}$ are then tied together by the interaction constructs. This is the same approach used in combining the set, bag, and list fragments of CPL into CPL; see Chapter 5. This approach to fusing languages sometimes leads to very interesting interaction operators. For example, when Libkin and I [131] glued the language of orsets and $\mathcal{NRA}$ into a single language, the orset-set interaction operator introduced is precisely the function which establishes the isomorphism between iterated powerdomains [128].

More rewrite rules

The identification of input token streams with output token streams gives rise to new possibilities: applying a scan construct to the output produced by a print construct and applying a putscan construct to the output of a put or a putscan construct. Fortunately, as demonstrated in the rewrite rules below, no new construct is required for optimization purpose.

- $scanObj(putObj\ e) \leadsto e$
- $scanObj(putPair(e_1, e_2)) \leadsto (scanObj\ e_1,\ scanObj\ e_2)$

Token Stream Monad

$$\frac{e_1 : [\![r]\!] \quad e_2 : [\![s \times t]\!]}{putscan\pi_1(e_1 \mid x^{[\![s]\!]} \lhd e_2) : [\![r]\!]} \qquad \frac{e_1 : [\![r]\!] \quad e_2 : [\![s \times t]\!]}{putscan\pi_2(e_1 \mid x^{[\![t]\!]} \lhd e_2) : [\![r]\!]}$$

$$\frac{e_1 : [\![s]\!] \quad e_2 : [\![t]\!]}{putPair(e_1, e_2) : [\![s \times t]\!]} \qquad \frac{}{putEmptySet^s : [\![\{s\}]\!]} \qquad \frac{e : [\![s]\!]}{putSingletonSet\ e : [\![\{s\}]\!]}$$

$$\frac{e_1 : [\![\{s\}]\!] \quad e_2 : [\![\{s\}]\!]}{putUnionSet(e_1, e_2) : [\![\{s\}]\!]} \qquad \frac{e_1 : [\![\{s\}]\!] \quad e_2 : [\![\{t\}]\!]}{putscanSet(e_1 \mid x^{[\![t]\!]} \lhd e_2) : [\![\{s\}]\!]}$$

Complex Object - Token Stream Interactions

$$\frac{e : s}{putObj\ e : [\![s]\!]} \qquad \frac{e_1 : r \quad e_2 : [\![s \times t]\!]}{scan\pi_1(e \mid x^{[\![s]\!]} \lhd e_2) : r} \qquad \frac{e_1 : r \quad e_2 : [\![s \times t]\!]}{scan\pi_2(e \mid x^{[\![t]\!]} \lhd e_2) : r}$$

$$\frac{e : [\![s]\!]}{scanObj\ e : s} \qquad \frac{e_1 : \{s\} \quad e_2 : [\![\{t\}]\!]}{scanSet(e_1 \mid x^{[\![t]\!]} \lhd e_2) : \{s\}} \qquad \frac{e_1 : [\![\{s\}]\!] \quad e_2 : \{t\}}{putSet(e_1 \mid x^t \in e_2) : [\![\{s\}]\!]}$$

Figure 6.5: The monad of token streams.

- $scanObj(putscan\pi_i(e_1 \mid x \lhd e_2)) \leadsto scan\pi_i(scanObj\ e_1 \mid x \lhd e_2)$
- $scanObj\ putEmptySet \leadsto \{\}$
- $scanObj(putSingletonSet\ e) \leadsto \{scanObj\ e\}$
- $scanObj(putUnionSet(e_1, e_2)) \leadsto (scanObj\ e_1) \cup (scanObj\ e_2)$
- $scanObj(putscanSet(e_1 \mid x \lhd e_2)) \leadsto scanSet(scanObj\ e_1 \mid x \lhd e_2)$
- $scanObj(putSet(e_1 \mid x \in e_2)) \leadsto \bigcup\{scanObj\ e_1 \mid x \in e_2\}$
- $scan\pi_i(e_1 \mid x \lhd putObj\ e_2) \leadsto e_1[putObj(\pi_i\ e_2)/x]$
- $scan\pi_i(e_1 \mid x \lhd putPair(e_2, e_3)) \leadsto e_1[e_i/x]$
- $scan\pi_i(e_1 \mid x \lhd putscan\pi_j(e_2 \mid y \lhd e_3)) \leadsto scan\pi_j(scan\pi_i(e_1 \mid x \lhd e_2) \mid y \lhd e_3)$
- $scanSet(e_1 \mid x \lhd putObj\ e_2) \leadsto \bigcup\{e_1[(putObj\ y)/x] \mid y \in e_2\}$
- $scanSet(e_1 \mid x \lhd\ putscan\pi_i(e_2 \mid y \lhd e_3)) \leadsto scan\pi_i(scanSet(e_1 \mid x \lhd e_2) \mid y \lhd e_3)$
- $scanSet(e \mid x \lhd putEmptySet) \leadsto \{\}$
- $scanSet(e_1 \mid x \lhd putSingletonSet\ e_2) \leadsto e_1[e_2/x]$
- $scanSet(e_1 \mid x \lhd putUnionSet(e_2, e_3)) \leadsto scanSet(e_1 \mid x \lhd e_2) \cup scanSet(e_1 \mid x \lhd e_3)$
- $scanSet(e_1 \mid x \lhd putscanSet(e_2 \mid y \lhd e_3)) \leadsto scanSet(scanSet(e_1 \mid x \lhd e_2) \mid y \lhd e_3)$
- $scanSet(e_1 \mid x \lhd putSet(e_2 \mid y \in e_3)) \leadsto \bigcup\{scanSet(e_1 \mid x \lhd e_2) \mid y \in e_3\}$
- $putscan\pi_i(e_1 \mid x \lhd putObj\ e_2) \leadsto e_1[(putObj(\pi_i\ e_2))/x]$
- $putscan\pi_i(e \mid x \lhd putPair(e_1, e_2)) \leadsto e_1[e_i/x]$
- $putscan\pi_i(e_1 \mid x \lhd putscan\pi_j(e_2 \mid y \lhd e_3))$
  $\leadsto putscan\pi_j(putscan\pi_i(e_1 \mid x \lhd xe_2) \mid y \lhd e_3)$
- $putscanSet(e \mid x \lhd putEmptySet) \leadsto putEmptySet$
- $putscanSet(e_1 \mid x \lhd putSingletonSet\ e_2) \leadsto e_1[e_2/x]$

- $putscanSet(e_1 \mid x \lhd putUnionSet(e_2, e_3))$
  $\leadsto putUnionSet(putscanSet(e_1 \mid x \lhd e_2), putscanSet(e_1 \mid x \lhd e_3))$
- $putscanSet(e_1 \mid x \lhd putscanSet(e_2 \mid y \lhd e_3))$
  $\leadsto putscanSet(putscanSet(e_1 \mid x \lhd e_2) \mid y \lhd e_3)$
- $putscanSet(e_1 \mid x \lhd putSet(e_2 \mid x \in e_3)) \leadsto putSet(putscanSet(e_1 \mid x \lhd e_2) \mid x \in e_3)$
- $putscanSet(e_1 \mid x \lhd putscan\pi_i(e_2 \mid y \lhd e_3))$
  $\leadsto putscan\pi_i(putscanSet(e_1 \mid x \lhd e_2) \mid y \lhd e_3)$

Rules such as $scanSet(e_1 \mid x \lhd putSet(e_2 \mid y \in e_3)) \leadsto \bigcup\{scanSet(e_1 \mid x \lhd e_2) \mid y \in e_3\}$ are optimization rules. Recall the $putSet(e_2 \mid y \in e_3)$ has a lazy semantics. Laziness comes with an overhead that can be costly. These rules turn the lazy constructs into equivalent eager ones, which are cheaper to execute.

Discussion

There are very few papers that explicitly considered the use of laziness in query processing and optimization. The only one that I know of is Buneman, Nikhil, and Frankel [30]. They were more concerned with reducing space consumption using laziness than in reducing response time. This bias is consistent with tradition. For the classical exposition of lazy evaluation stressed the possibility of using lazy evaluation to explore infinite search space (the sieve of Eratosthenes being a favourite example; see Field and Harrison [66]). This tradition accentuated the space-saving virtue of lazy evaluation, while response-time improvement had not been emphasized.

The usual way to build compilers for lazy languages such as Haskell [62] and Lazy ML [12] is to evaluate everything lazily by default. Then, use sophisticated techniques [24, 5] to perform strictness analysis on programs to bring in eagerness. My emphasis on using lazy evaluation to improve response time makes my approach different. I execute everything eagerly by default. Then I use the simple rewrite rules presented above to introduce laziness into my programs in a profitable way.

I emphasize again that my token stream monad constructs closely correspond to the constructs for my nested relational calculus. This correspondence is not surprising because both are organized around the categorical concept of a monad and both are obtained by turning universal properties into syntax. The fusion of the physcial language and the abstract language is cleanly obtained via the complex object and token stream interaction constructs. These interaction constructs are what Wadler called monad morphisms [198]. This design principle is the cohesive thread that links together the concrete language CPL, the abstract language $\mathcal{NRC}$, and the physical language of token streams.

The recent work of Fegaras [64] is closely related to the work here. His paper is influenced by the work of Buneman, Ohori, Tannen, Wadler, and myself [29, 198, 204, 32, 155]. He independently found that abstract programming constructs on collection types can be mapped to physical programming constructs having the same form. While he gave a good treatment of the mapping from abstract constructs to physical constructs, he did not consider the significance of alternative operational semantics and he did not provide specific rules for mapping from physical constructs to specific algorithms. For example, a direct interpretation of his merge join program is still a nested loop. My treatment is more in depth in both cases because the impact of laziness on performance is considered in this chapter and specific rules for mapping to joins algorithms are given in the next chapter.

Chapter 7

Additional Optimizations

Instruction for reading: Skip. PHILIP JOHNSON-LAIRD

There exists a large body of literature on physical optimization in flat relational systems. See Graefe [75]; Jarke and Koch [109]; Kim [115]; Mishra and Eich [145]; Nakayama, Kitsuregawa, and Takagi [148]; Selinger, Astrahan, Chamberlin, Lorie, and Price [171]; etc. This chapter applies some of the better known traditional optimization techniques to $\mathcal{NRC}$. Even though these techniques are not new, I think this chapter is a contribution in at least two ways. Flat relational systems deal in an impoverished class of data that excludes nested relations but $\mathcal{NRC}$ deals with a much richer class of data that includes nested relations. Hence the application of these techniques to $\mathcal{NRC}$ is also a generalization of these techniques. Flat relational systems are generally implemented by a collection of enriched operators based upon the flat relational algebra but $\mathcal{NRC}$, as seen in Chapter 6, is implemented on top of operators based upon the categorical notion of a monad. Hence the application of these techniques to $\mathcal{NRC}$ is also a demonstration that my "monadic" framework does not obstruct techniques conceived in an alient way.

I stress that the rewrite rules presented in this chapter are not intended to be complete. Rather, they are intended to give a taste of how less tidy optimization techniques can be added to my system. I also assume the use of the commutative rule

if e_1 *then* (*if* e_2 *then* e_3 *else* e_4) *else* $e_4 \rightsquigarrow$ *if* e_2 *then* (*if* e_1 *then* e_3 *else* e_4) *else* e_4 throughout this chapter.

Organization

Section 7.1. Two new constructs are introduced. The first is for caching small external relations into memory. The second is for indexing small external relations into memory. Some rules are suggested for using these new operators in query optimization.

Section 7.2. A new construct is introduced to capture the blocked nested-loop join algorithm. Some rules for using this operator in query optimization are presented; in particular rules for recognizing a nested loop to be a join are given.

Section 7.3. A new construct is introduced to capture the indexed blocked-nested-loop join algorithm. Some rules for using this operator in query optimization are presented; in particular rules for recognizing whether the join condition in a blocked nested-loop join can be dynamically indexed or not are given.

Section 7.4. A new construct is introduced for caching large intermediate results on disk to avoid recomputation. Some rules for using this operator in query optimization are presented.

Section 7.5. A new construct is introduced to illustrate the use of relational servers as providers of external data. Some rules for migrating queries to such servers are presented. In particular, rules for the migration of selection, projection, and join operations are illustrated.

Section 7.6. A new construct is introduced to illustrate the use of nonrelational servers as providers of external data. Some rules for migrating queries to such servers are presented. In particular, rules for moving selection and set-flattening operations are given.

7.1 Caching and indexing small relations

Using the techniques of Chapter 6, joins in $\mathcal{NRC}$ are expressed in the physical language using nested loops of the form $putscanSet(putscanSet(\mathit{if}\ p(x,y)\ \mathit{then}\ q(x,y)\ \mathit{else}\ putEmptySet \mid y \lhd R) \mid x \lhd S)$. Evaluating this program causes the inner relation R to be fetched from disk (or worse — brought in from a slow remote site) into memory as many times as there are tuples in the outer relation S. However, if R is small enough to fit completely into the available memory, then such repeated fetching can be avoided. The first half of this section considers the general situation where nothing is known about the join condition p. The second half of this section considers the special situation where the join condition p involves an equality test, which can be turned into an index probe.

Caching small relations

The new construct in Figure 7.1 is introduced to achieve the effect of caching small relations in a general way. Semantically, $cache(e_1, e_2) = e_2()$. That is, $cache(e_1, e_2)$ is required to

$$\frac{e_1 : \mathbb{N} \quad e_2 : unit \to [\![\{s\}]\!]}{cache(e_1, e_2) : [\![s]\!]}$$

Figure 7.1: The construct for caching small relations.

return the same result as $e_2()$. However, $cache(e_1, e_2)$ is given an operational semantics with the following side effect. The first time $cache(e_1, e_2)$ is invoked during query evaluation, a cache is created. This cache is identified by the natural number e_1 and the token stream $e_2()$ is stored in the cache. The token stream $e_2()$ is then returned as the result. The next time $cache(e_1, e_2)$ is invoked, the token stream already stored in the cache identified by e_1 is directly returned.

Notice that the operational semantics of $cache(e_1,\ e_2)$ is not sound with respect to the equation $cache(e_1, e_2) = e_2()$. To achieve soundness, it is sufficient to impose three conditions on $cache(e_1, e_2)$. The first condition is that e_2 should be a constant identifying an external data source. The second condition is that e_1 is required to be a constant. The third condition is that if $cache(e_1, e_2)$ and $cache(e_1', e_2')$ occur in two places in a query, then e_1 and e_1' must be different unless e_2 and e_2' are identical. These conditions are easily guaranteed if the *cache* construct is only introduced during optimization and is not present in the original query.

The basic optimization rule to exploit this construct is

- $R \rightsquigarrow cache(n, \lambda x.R)$, if $R : [\![s]\!]$ is an identifer of an external data source (for example, a file pointer); the size of R is determined to be small enough to fit into memory; and n is a fresh cache identifier.

The effectiveness of this rule is easily illustrated. Consider the query $scanSet(scanSet(e \mid x \lhd R) \mid y \lhd S)$ where R is a small external relation and S a big relation. Then R has to be scanned as many times as there are objects in S. Using the rule above, the query is rewritten to $scanSet(scanSet(e \mid x \lhd cache(1, \lambda z.R)) \mid y \lhd S)$. Thus R is scanned only once. the improvement in total time is obvious.

Indexing small relations

The new construct in Figure 7.2 is introduced to achieve the effect of indexing a small relation. Semantically, $index(e_1, e_2, e_3)(o) = scanSet(\mathit{if}\ e_3(scanObj\ x) = o\ \mathit{then}\ \{scanObj\ x\}\ \mathit{else}\ \{\} \mid x \lhd e_2())$. That is, it returns all members of $e_2()$ having an index value equals to o. However, $index(e_1, e_2, e_3)$ is given an operational semantics with the following side effect. The first time it is executed, an indexed cache is created. The cache is identified by the natural number e_1. The index function to be associated with the indexed cache is the function e_3. The complex object O represented by the token stream $e_2()$ is stored in the indexed cache. The index key of an item is obtained by applying e_3 to that

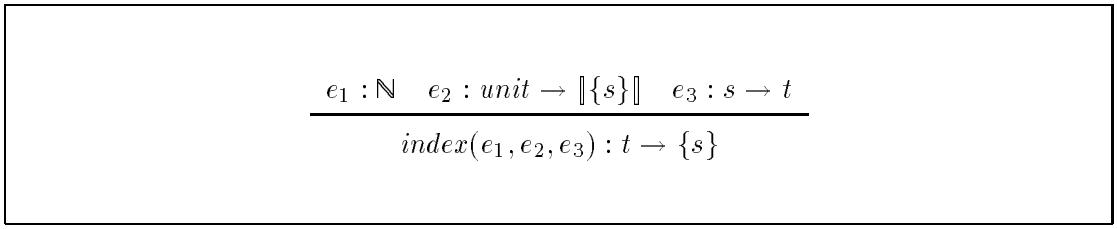


Figure 7.2: The construct for indexing small relations.

item. Note that several elements may map via e_3 to the same bucket in the cache. The function $f(o) = \{o' \mid o' \in O,\ e_3(o) = e_3(o')\}$ is returned. (This function is implemented by applying e_3 to the input o to obtain the index key, which can then be used to access the indexed cache to bring out the bucket containing all the matching items.) The next time $index(e_1, e_2, e_3)$ is executed, the function f is directly returned.

Observe that the operational semantics of $index(e_1, e_2, e_3)$ is not sound with respect to its intended equational theory in general. To achieve soundness, it is sufficient to impose four conditions on $index(e_1, e_2, e_3)$. The first condition is that e_2 should be a constant identifying an external data source. The second condition is that e_1 should be a constant. The third condition is that e_3 should have not free variable. The fourth condition is that whenever $index(e_1, e_2, e_3)$ and $index(e_1', e_2', e_3')$ appear in two places in a query, then e_1 and e_1' must be distinct unless e_2 and e_2' are identical and e_3 and e_3' are identical. These conditions are easily arranged if the *index* construct is only introduced during optimization.

The basic optimization rules to exploit this construct are given below. They essentially check if an equality test can be turned into index probe. These two rules are built on top of the rule introduced earlier for $cache(e_1, e_2)$. I prefer building my optimization rules in this incremental fashion. It is my experience that doing so greatly reduces the number of optimization rules in my system.

- $putscanSet(if\ e_1 = e_2\ then\ e_3\ else\ putEmptySet \mid x \lhd cache(n, e_4)) \rightsquigarrow putSet($ $e_3[(putObj\ y)/x] \mid y \lhd index(m, e_4, \lambda z.e_1[(putObj\ z)/x])(e_2))$, if m is fresh, x is the only free variable in e_1, and x is not free in e_2.

- $putscanSet(if\ e_1 = e_2\ then\ e_3\ else\ putEmptySet \mid x \lhd cache(n, e_4)) \rightsquigarrow putSet(e_3[(putObj\ y)/x] \mid y \lhd index(m, e_4, \lambda z.e_2[(putObj\ z)/x])(e_1))$, if m is fresh, x is the only free variable in e_2, and x is not free in e_1.

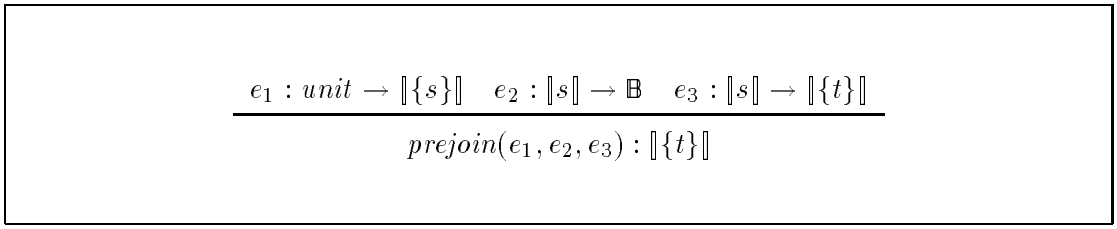
The effectiveness of these rules is easily illustrated. Consider the query $putscanSet(putscanSet(if\ f(x) = g(y)\ then\ e\ else\ putEmptySet \mid x \lhd R) \mid y \lhd S)$ where R is a small external relation and S a big relation. Then R has to be loaded as many times as there are elements in S and the equality test has to be performed a quadratic number of times. It is rewritten to $putscanSet(putSet(e[(putObj\ z)/x] \mid z \lhd index(m,\ \lambda u.R,\ \lambda v.f(putObj\ v))(g\ y)) \mid y \lhd S)$. Then R is loaded once and the equality test is performed quasi-linear number of times. The improvement in total time is obvious.

7.2 Blocked nested-loop join

One of the earliest method for improving performance of joins is the blocked nested-loop technique [115]. The basic idea is to divide the inner and outer relations into blocks, each of which is small enough to fit into memory. Then perform the join by joining each block of the inner relation with each block of the outer relation using any efficient main memory technique. Using the technique, the inner relation is scanned as many times as there are blocks, as opposed to records, in the outer relation. (Further improvement can be gained by scanning boustrophedonically. That is, the direction of scanning for one of the relations is alternated so that the last block read in each direction need not be re-scanned when the direction is changed. See Kim [115].) This technique is applicable even when the join condition is not an equality test. It is a generalization of the caching technique to inner relations that are too big to be cached entirely in memory.

Preparing a join

To simplify subsequent analysis, the new primitive in Figure 7.3 is introduced. In every



$$\frac{e_1 : unit \to [\![\{s\}]\!] \quad e_2 : [\![s]\!] \to \mathbb{B} \quad e_3 : [\![s]\!] \to [\![\{t\}]\!]}{prejoin(e_1, e_2, e_3) : [\![\{t\}]\!]}$$

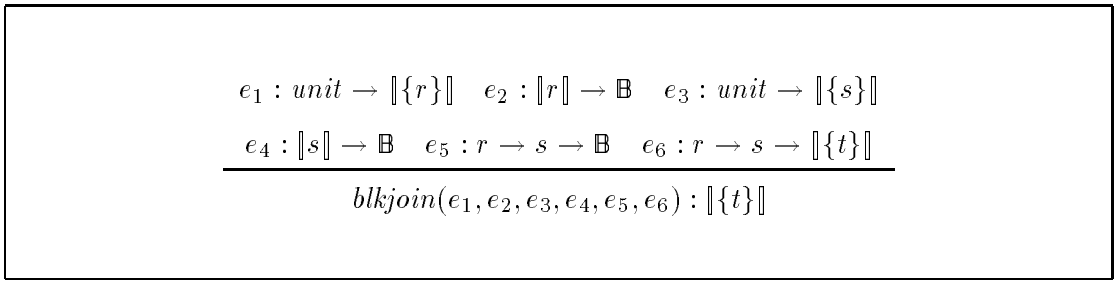
Figure 7.3: The construct for a filter loop.

way, $prejoin(e_1, e_2, e_3) = putscanSet(\mathit{if}\ e_2(x)\ \mathit{then}\ e_3(x)\ \mathit{else}\ putEmptySet \mid x \lhd e_1())$. It is used in conjunction with the basic rules below for putting queries into a simpler form for subsequent analysis.

- $putscanSet(e_1 \mid x \lhd e_2) \rightsquigarrow prejoin(\lambda y.e_2,\ \lambda z.true,\ \lambda x.e_1)$
- $prejoin(e_1,\ e_2,\ \lambda x.\mathit{if}\ e_3\ \mathit{then}\ e_4\ \mathit{else}\ putEmptySet) \rightsquigarrow prejoin(e_1,\ \lambda x.\mathit{if}\ e_2(x)\ \mathit{then}\ e_3\ \mathit{else}\ \mathit{false}, \lambda x.e_4)$, if x is the only free variable in e_3.

Blocked nested-loop join

The new construct in Figure 7.4 is needed to capture the blocked nested-loop join algorithm. In terms of semantics, $blkjoin(e_1, e_2, e_3, e_4, e_5, e_6) = putscanSet(\mathit{if}\ e_2(x)\ \mathit{then}\ putscanSet($



$$\frac{\begin{array}{c} e_1 : unit \to [\![\{r\}]\!] \quad e_2 : [\![r]\!] \to \mathbb{B} \quad e_3 : unit \to [\![\{s\}]\!] \\ e_4 : [\![s]\!] \to \mathbb{B} \quad e_5 : r \to s \to \mathbb{B} \quad e_6 : r \to s \to [\![\{t\}]\!] \end{array}}{blkjoin(e_1, e_2, e_3, e_4, e_5, e_6) : [\![\{t\}]\!]}$$

Figure 7.4: The construct for blocked nested-loop join.

$\mathit{if}\ e_4(y)\ \mathit{then}\ \mathit{if}\ e_5(scanObj\ x)(scanObj\ y)\mathit{then}\ e_6(scanObj\ x)(scanObj\ y)\ \mathit{else}\ putEmptySet\ \mathit{else}\ putEmptySet \mid y \lhd e_3())\ \mathit{else}\ putEmptySet \mid x \lhd e_1())$. In other words, $e_1()$ is the

outer relation of the join, e_2 is the selection predicate on the outer relation, $e_3()$ is the inner relation of the join, e_4 is the selection predicate on the inner relation, e_5 is the join condition, and e_6 is the transformation to be applied to qualified records. This primitive is implemented using the blocked nested-loop join algorithm.

Some of the basic rules for exploiting this new construct are given below. The first rule recognizes the basic opportunity for a blocked nested-loop join. The second rule recognizes the occurrence of a qualification test in the transformer part of the join and combines it with the join condition. The third and fourth rules detect that certain parts of the join condition involve only the inner record or the outer record and combine these tests with the inner predicate and outer predicates respectively. The remaining rules handle some of the possible interactions between *prejoin* and *blkjoin*.

- $prejoin(e_1,\ e_2,\ \lambda x.prejoin(e_3, e_4, e_5)) \rightsquigarrow blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.true, \lambda y.\lambda z.\ e_5[(putObj\ y)/x]\ (putObj\ z))$, if x is not free in e_3 and e_4.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda y.\lambda z.if\ e_6\ then\ e_7\ else\ putEmptySet) \rightsquigarrow blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.if\ e_5(y)(z)\ then\ e_6\ else\ false,\ \lambda y.\lambda z.e_7)$

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.if\ e_5\ then\ e_6\ else\ false,\ e_7) \rightsquigarrow blkjoin(e_1,\ e_2,\ e_3,\ \lambda v.if\ e_4(v)\ then\ e_5[(scanObj\ v)/z]\ else\ false,\ \lambda y.\lambda z.e_6,\ e_7)$, if y is not free is e_5.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.if\ e_5\ then\ e_6\ else\ false,\ e_7) \rightsquigarrow blkjoin(e_1,\ \lambda u.if\ e_2(u)\ then\ e_5[(scanObj\ u)/y]\ else\ false,\ e_3,\ e_4,\ \lambda y.\lambda z.e_6,\ e_7)$, if z is not free in e_5.

- $prejoin(e_1,\ e_2,\ \lambda x.blkjoin(e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8)) \rightsquigarrow blkjoin(e_1,\ e_2,\ \lambda u.blkjoin(\ e_3,\ e_4,\ e_5,\ e_6,\ e_7, \lambda y.\lambda z.putObj\{(y,z)\}),\ \lambda z.true,\ \lambda y.\lambda z.true,\ \lambda y.\lambda z.(\lambda x.e_8)\ (putObj\ y)(\pi_1\ z)(\pi_2\ z))$, if x is not free in e_3, e_4, e_5, e_6, and e_7.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda y.\lambda z.prejoin(e_6,\ e_7,\ e_8)) \rightsquigarrow prejoin(e_6,\ e_7,\ \lambda x.blkjoin(\ e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.\lambda z.e_8(x)))$.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda y.\lambda z.blkjoin(e'_1,\ e'_2,\ e'_3,\ e'_4,\ e'_5,\ e)) \rightsquigarrow blkjoin(\ \lambda x.blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda u.\lambda v.putObj\{(u,v)\}),\ \lambda u.true,\ \lambda x.blkjoin(e'_1,\ e'_2,\ e'_3,$

$e'_4,\ e'_5,\ \lambda u.\lambda v.putObj\{(u,v)\}),\ \lambda v.true,\ \lambda u.\lambda v.true,\ \lambda u'.\lambda v'.(\lambda y.\lambda z.e)(\pi_1\ u')(\pi_2\ u')$ $(\pi_1\ v')(\pi_2\ v'))$, if y and z are not free in e'_1, e'_2, e'_3, e'_4, and e'_5.

- $prejoin(\lambda u.blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6),\ e_7,\ e_8) \rightsquigarrow blkjoin(e1,\ e_2,\ e_3,\ e_4,\ e_5,$ $\lambda y.\lambda z.putscanSet(if\ e_7(x)\ then\ e_8(x)\ else\ putEmptySet \mid x \lhd e_6(y)(z)))$, if u not free in e_1, e_2, e_3, e_4, e_5, and e_6.

The effectiveness of these rules is easily illustrated. Consider the query $putscanSet($ $putscanSet(if\ f(x) = g(y)\ then\ e\ else\ putEmptySet \mid x \lhd R) \mid y \lhd S)$, where both R and S are too big to fit in memory. Then R has to be loaded as many times as there are elements in S. It is rewritten to $blkjoin(\lambda u.S,\ \lambda u.true,\ \lambda v.R,\ \lambda v.true,$ $\lambda u.\lambda v.f(putObj\ v) = g(putObj\ u),\ \lambda u.\lambda v.e[(putObj\ u)/y, (putObj\ v)/x])$. Then R is loaded as many times as there are blocks in S. The performance is improved by a factor proportional to the blocking factor used.

7.3 Indexed blocked-nested-loop join

Suppose the join condition involves an equality test of the form $f(x) = g(y)$ where x is bound in the outer relation and y to the inner relation. Then it is possible to dynamically create an index for the outer relation using f as the indexing function and g as the probe function. The blocked nested-loop join can be turned into the indexed blocked-nested-loop join by taking advantage of such special join conditions.

The basic idea is similar to the dynamic staging and hashing idea of Nakayama, Kitsuregawa, and Takagi [148]. Divide the outer relation into blocks. Bring one of these blocks into memory. Dynamically index it. Join this indexed block with the inner relation using any efficient main memory indexed join algorithm. Repeat the previous steps for the remaining blocks of the outer relation. Note that each block is small enough so that the index created dynamically for it can fit into memory. Using this technique, the outer relation is scanned only once, while the inner relation is scanned as many times as there are blocks in the outer relation. Furthermore, the join condition is computed only a quasi-linear number times, as

opposed to a quadratic number of times.

The new construct in Figure 7.5 is needed to capture the indexed blocked-nested-loop join algorithm. In terms of semantics, $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8)$ =

$$\frac{\begin{array}{c} e_1 : unit \rightarrow [\![\{r\}]\!] \quad e_2 : [\![r]\!] \rightarrow \mathbb{B} \quad e_3 : r \rightarrow s \\ e_4 : unit \rightarrow [\![\{t\}]\!] \quad e_5 : [\![t]\!] \rightarrow \mathbb{B} \quad e_6 : t \rightarrow s \\ e_7 : r \rightarrow t \rightarrow \mathbb{B} \quad e_8 : r \rightarrow t \rightarrow [\![\{u\}]\!] \end{array}}{idxjoin(e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8) : [\![\{u\}]\!]}$$

Figure 7.5: The construct for indexed blocked-nested-loop join.

$putscanSet(\ if\ e_2(y)\ then\ putscanSet(\ if\ e_5(z)\ then\ if\ e_3(scanObj\ y) = e_6(scanObj\ z)\ then\ if\ e_7(scanObj\ y)(scanObj\ z)\ then\ e_8(scanObj\ y)(scanObj\ z)\ else\ putEmptySet\ else\ putEmptySet\ else\ putEmptySet \mid z \lhd e_4())\ else\ putEmptySet \mid y \lhd e_1())$. In other words, $e_1()$ is the outer relation of the join, e_2 is the selection predicate on the outer relation, e_3 is the indexing function, $e_4()$ is the inner relation, e_5 is the selection predicate on the inner relation, e_6 is the probe function, e_7 is the join condition, and e_8 is the transformation to be applied to qualified records. This primitive is implemented using the indexed blocked-nested-loop join algorithm.

Some of the basic rules for exploiting this operator are given below. The first and second rules recognize the basic opportunity for an indexed blocked-nested-loop join. The third rule discovers that the transformer of the join contains a test which can be combined with the join condition. The fourth and fifth rules recognize that parts of the join condition can be combined with the outer or the inner predicates of the join. The sixth and seventh rules recognize that the join condition contains an equality test that can be turned into an index probe and proceed to do so. The remaining rules are a sampling of the ways in which *prejoin*, *blkjoin*, and *idxjoin* interact.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.if\ e_5 = e_6\ then\ e_7\ else\ false,\ e_8) \rightsquigarrow idxjoin(e_1,\ e_2,\ \lambda y.e_5,\ e_3,\ e_4,\ \lambda z.e_6,\ \lambda y.\lambda z.e_7,\ e_8)$, if y is the only free variable in e_5 and z is the only free variable in e_6.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda y.\lambda z.if\ e_5 = e_6\ then\ e_7\ else\ false,\ e_8) \rightsquigarrow idxjoin(e_1,\ e_2,\ \lambda y.e_6,\ e_3,\ e_4,\ \lambda z.e_5,\ \lambda y.\lambda z.e_7,\ e_8)$, if y is the only free variable in e_6 and z is the only free variable in e_5.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.if\ e_8\ then\ e_9\ else\ putEmptySet) \rightsquigarrow idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.\lambda z.if\ e_8\ then\ e_7(y)(z)\ else\ false,\ \lambda y.\lambda z.e_9)$

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.\lambda z.if\ e_7\ then\ e_8\ else\ false,\ e_9) \rightsquigarrow idxjoin(e_1,\ \lambda u.if\ e_2(u)\ then\ e_7[(scanObj\ u)/y]\ else\ false,\ e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.\lambda z.e_8,\ e_9)$, if z is not free in e_7.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.\lambda z.if\ e_7\ then\ e_8\ else\ false,\ e_9) \rightsquigarrow idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ \lambda u.if\ e_5(u)\ then\ e_7[(scanObj\ u)/z]\ else\ false,\ e_6,\ \lambda y.\lambda z.e_8,\ e_9)$, if y is not free in e_7.

- $idxjoin(e_1,\ e_2,\ e_3,e_4,\ e_5,e_6,\ \lambda y.\lambda z.if\ e_7 = e_8\ then\ e_9\ else\ false,\ e) \rightsquigarrow idxjoin(e_1,\ e_2,\ \lambda y.(e_3(y),e_7),\ e_4,\ e_5,\ \lambda z.(e_6(z),e_8),\ \lambda y.\lambda z.e_9,\ e)$, if z is free in e_8 but not in e_7 and y is free in e_7 but not in e_8.

- $idxjoin(e_1,\ e_2,\ e_3,e_4,\ e_5,e_6,\ \lambda y.\lambda z.if\ e_7 = e_8\ then\ e_9\ else\ false,\ e) \rightsquigarrow idxjoin(e_1,\ e_2,\ \lambda y.(e_3(y),e_8),\ e_4,\ e_5,\ \lambda z.(e_6(z),e_7),\ \lambda y.\lambda z.e_9,\ e)$, if y is free in e_8 but not in e_7 and z is free in e_7 but not in e_8.

- $prejoin(e_1,\ e_2,\ \lambda x.idxjoin(e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8,\ e_9,\ e)) \rightsquigarrow blkjoin(e_1,\ e_2,\ \lambda v.idxjoin(e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8,\ e_9,\ \lambda y.\lambda z.putObj\{(y,z)\}),\ \lambda v.true,\ \lambda y.\lambda z.true,\ \lambda y.\lambda x'.e[(putObj\ y)/x](\pi_1\ x')(\pi_2\ x'))$, if x is not free in e_3, e_4, e_5, e_5, e_7, e_8, and e_9.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.prejoin(e_8,\ e_9,\ e)) \rightsquigarrow prejoin(e_8,\ e_9,\ \lambda x.idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.e\ x))$, if y and z are not free in e_8 and e_9.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.idxjoin(e_1',e_2',\ e_3',e_4',\ e_5',\ e_6',\ e_7',\ e)) \rightsquigarrow blkjoin(\lambda u.idxjoin(\ e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.putObj\{(y,\ z)\}),\ \lambda u.true,$

$\lambda v.idxjoin(e'_1,\ e'_2,\ e'_3,\ e'_4,\ e'_5,\ e'_6,\ e'_7,\ \lambda y.\lambda z.putObj\{(y,z)\}),\ \lambda v.true,\ \lambda u.\lambda v.true,$ $\lambda y'.\lambda z'.(\lambda y.\lambda z.e(\pi_1\ z')(\pi_2\ z'))(\pi_1\ y')(\pi_2\ y'))$, if y and z are not free in e'_1, e'_2, e'_3, e'_4, e'_5, e'_6, and e'_7.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda y.\lambda z.idxjoin(e'_1,\ e'_2,\ e'_3,\ e'_4,\ e'_5,\ e'_6,\ e'_7,\ e)) \rightsquigarrow blkjoin($ $\lambda u.blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ \lambda y.\lambda z.putObj\{(y,\ z)\}),\ \lambda u.true,\ \lambda v.idxjoin(e'_1,\ e'_2,\ e'_3,$ $e'_4,\ e'_5,\ e'_6,\ e'_7,\ \lambda y.\lambda z.putObj\{(y,\ z)\}),\ \lambda v.true,\ \lambda u.\lambda v.true,\ \lambda y'.\lambda z'.(\lambda y.\lambda z.e(\pi_1\ z')(\pi_2$ $z'))(\pi_1\ y')(\pi_2\ y'))$, if y and z are not free in e'_1, e'_2, e'_3, e'_4, e'_5, e'_6, and e'_7.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.blkjoin(e'_1,\ e'_2,\ e'_3,\ e'_4,\ e'_5,\ e)) \rightsquigarrow blkjoin(e'_1,$ $e'_2,\ e'_3,\ e'_4,\ e'_5,\ \lambda y'.\lambda z'.idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.e(y')(z')))$, if y and z are not free in e'_1, e'_2, e'_3, e'_4, and e'_5.

- $prejoin(\lambda u.idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8),\ e_9, e) \rightsquigarrow idxjoin(e_1,\ e_2,\ e_3,$ $e_4,\ e_5,\ e_6,\ e_7,\ \lambda y.\lambda z.putSet(if\ e_9(x)\ then\ e(x)\ else\ putEmptySet \mid x \lhd e_8(y)(z)))$, if u is not free in e_1, e_2, e_3, e_4, e_5, e_6, e_7, and e_8.

The effectiveness of these rules is easily illustrated. Consider the query $putscanSet($ $putscanSet(if\ f(x) = g(y)\ then\ e\ else\ putEmptySet \mid x \lhd R) \mid y \lhd S)$, where both R and S are too big to fit in memory. Then R has to be loaded as many times as there are elements in S. Furthermore, the equality test has to be performed $m \cdot n$ times where m and n are the cardinalities of R and S. It is rewritten to $idxjoin(\lambda u.S,\ \lambda u.true,\ \lambda u.g(putObj\ u),\ \lambda v.R,\ \lambda v.true,\ \lambda v.f(putObj\ v),\ \lambda u.\lambda v.true,\ \lambda u.\lambda v.$ $e[(putObj\ u)/y, (putObj\ v)/x])$. Then S is loaded once, a block at a time; R is loaded as many times as there are blocks in S; and the equality test is performed $m \cdot \log n$ times.

7.4 Caching inner relations

The inner relation in a join may not be a base table. It can be a subquery. Under such a situation, this subquery may have to be recomputed several times. For example, if *blkjoin* or *idxjoin* is used to evaluate the join, then the inner subquery has to be recomputed as

many times as there are blocks in the outer relation. The optimizations suggested so far do not consume disk storage other than that needed to store the input to and the output of a query. By allowing additional disk storage to be used, large intermediate data can be cached to avoid recomputation.

The new construct in Figure 7.6 is introduced to achieve the effect of caching the result of subqueries on disk. Semantically, $bigcache(e_1, e_2) = e_2()$. That is, $bigcache(e_1, e_2)$ is

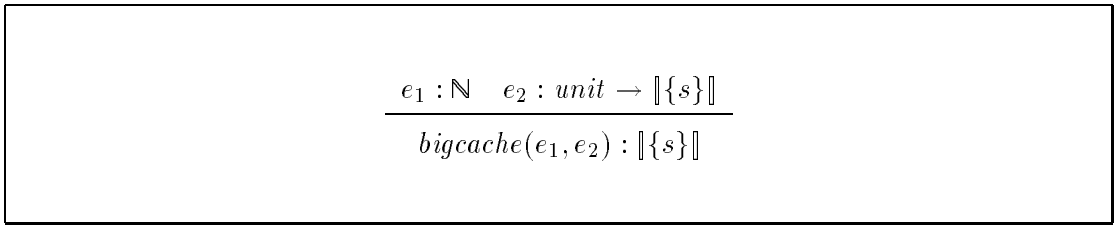


Figure 7.6: The construct for caching large intermediate results on disk.

required to return the same result as $e_2()$. However, $bigcache(e_1, e_2)$ is given an operational semantics with the following side effect. The first time $bigcache(e_1, e_2)$ is invoked during query evaluation, a file is created on disk. This file is identified by the natural number e_1 and the token stream $e_2()$ is written to that file. The file is then returned as the result. The next time $bigcache(e_1, e_2)$ is invoked, the file is directly returned, without recomputing $e_2()$.

The operational semantics of $bigcache(e_1, e_2)$ is not sound with respect to the equation $bigcache(e_1, e_2) = e_2()$. To achieve soundness, it is sufficient to impose three conditions on $bigcache(e_1, e_2)$. The first condition is that e_2 should have no free variable. The second condition is that e_1 should be a constant. The third condition is that if $bigcache(e_1, e_2)$ and $bigcache(e_1', e_2')$ occur in two places in a query, then e_1 and e_1' must be different unless e_2 and e_2' are identical. These conditions are easy to ensure if the *bigcache* is only brought in during optimization and is not present in the original query.

The basic optimization rules to exploit this construct are given below. They basically recognized that the inner relation of a join is not a base table and is not yet cached.

- $blkjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6) \leadsto blkjoin(e_1,\ e_2,\ \lambda v.bigcache(n,\ \lambda v.putscanSet($ $if\ e_4(x)\ then\ putSingletonSet(x)\ else\ putEmptySet \mid x \triangleleft e_3())),\ \lambda v.true,\ e_5,\ e_6)$, if n is fresh, e_4 and e_3 have no free variables, e_3 is not of the form $\lambda v.bigcache(m, e)$, and e_3 is not a base table.

- $idxjoin(e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8) \leadsto idxjoin(e_1,\ e_2,\ e_3,\ \lambda v.bigcache(n,$ $\lambda v.putscanSet(if\ e_5(x)\ then\ putSingletonSet(x)\ else\ putEmptySet \mid x \triangleleft e_4())),$ $\lambda u.true,\ e_6,\ e_7,\ e_8)$, if n is fresh, e_4, e_5, and e_6 have no free variable, e_4 is not of the form $\lambda v.bigcache(m, e)$, and e_4 is not a base table.

Consider the query $putscanSet(putscanSet(if\ f(x) = g(y)\ then\ e\ else\ putEmptySet \mid x \triangleleft R) \mid y \triangleleft S)$, where both R and S are too big to fit in memory. Suppose R is a subquery, as opposed to a base table. Then R has to be recomputed and loaded as many times as there are elements in S. It is rewritten to $idxjoin(\lambda u.S,\ \lambda u.true,\ \lambda u.g(putObj\ u),$ $\lambda v.bigcache(n,\ \lambda v.R),\ \lambda v.true,\ \lambda v.f(putObj\ v),\ \lambda u.\lambda v.true,\ \lambda u.\lambda v.e[(putObj\ u)/y, (putObj$ $v)/x])$. Then S is loaded once, a block at a time; R is computed once, the result is cached on disk and loaded as many times as there are blocks in S. If recomputation of R is costly, then the improvement of this optimization is significant.

7.5 Pushing operations to relational servers

Suppose some of the input to a query comes from an external data source that has some query processing capabilities. For example, the input might actually be produced by a full-fledged relational server. It is usually profitable to migrate some manipulation of this input to the source server. The first advantage is the reduction of the load on the local machine. The second advantage is that the source server usually has further information, such as existence of precomputed indices or frequency statistics, which is useful for improved optimization. The third advantage is that several source servers can be kept busy simultaneously and thus increase parallelism.

This section outlines how a relational server can be exploited. For simplicity of presentation,

a single server is assumed here. It is straightforward to generalize this optimization to multiple servers; see Chapter 9. The new construct in Figure 7.7 is needed. where e is an

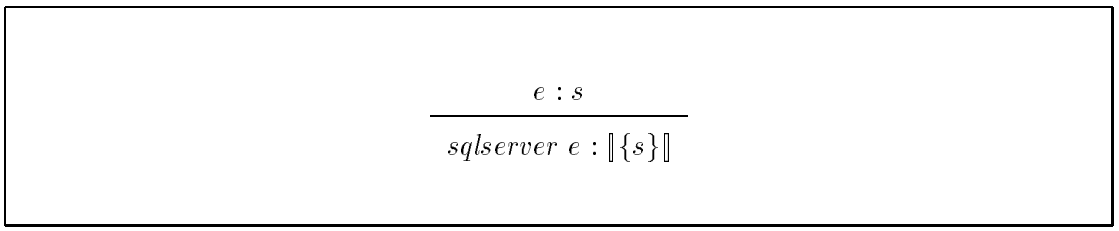


Figure 7.7: The construct for accessing a relational server.

expression that can be translated into an SQL query of the form select COLUMNS from TABLES where CONDITIONS. COLUMNS are restricted to simple label names qualified by table names or is the wildcard character *. TABLES are restricted to relations stored on that server. CONDITIONS are simple conjunctions of equality and inequality tests. For simplicity here, I directly write this e as the string select COLUMNS from TABLES where CONDITIONS. (This simplification is actually not far from my implementation. See the sample optimizer output scripts in Section 8.5 and see Chapter 9.)

Let me give a couple of short examples to illustrate this construct. The query *sqlserver* select * from locus where 1 = 1 fetches the table locus from the server and brings it to the local system. The query *sqlserver* select gb-head-accs.locus, gb-head-accs.accession, gb-head-accs.title, gb-head-accs.length, gb-head-accs.taxname from gb-head-accs where gb-head-accs.pastaccession = "M15492" selects the specified fields from records in gb-head-accs having pastaccession of M15492 from the server and brings it to the local system.

One more note before I plunge into the details of optimization. Relations are really sets of records. So I follow CPL and use records instead of pairs in this section. Records are formed by $(l_1 : e_1, \ldots, l_n : e_n)$ and fields are selected by $\pi_{l_i}\ e$ where l_i are labels. Analogous operations on token streams are used as well.

Some basic rules are given below. Throughout these rules, I assumed that the table names in TABLES have been properly aliased to avoid name clashes. The first five rules show how

to move selection predicates to the server. I use the equality test as the example predicate to be migrated. Other predicates, such as $\leq$, $\geq$, $<$, and $>$, can be similarly shifted to the server; the only requirement is that they be simple enough for the server. The next five rules give an idea of how to move projection operations to the server. The last three rules illustrate how to move joins to the server. I use equality test as the example join predicate to be migrated. Other join predicates, such as $\leq$, $\geq$, $<$, and $>$, can be similarly shifted to the server as long as they are simple enough for the server.

- $prejoin(\lambda x.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda x.if\ c = scan\pi_l(scanObj\ y \mid y \lhd x)\ then\ e_2\ else\ false,\ e_3) \rightsquigarrow prejoin(\lambda x.sqlserver$ select COLUMNS from TABLES where CONDITIONS and t.l = c, $\lambda x.e_2,\ e_3)$, if c is a constant, l is a column name qualified by table name t in COLUMNS, or TABLES consists of just the table name t.

- $blkjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda u.if\ c = scan\pi_l(\ scanObj\ y \mid y \lhd u)\ then\ e_2\ else\ false,\ e_3,\ e_4,\ e_5,\ e_6) \rightsquigarrow blkjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS and t.l = c, $\lambda u.e_2,\ e_3,\ e_4,\ e_5,\ e_6)$, if c is a constant, l is a column name qualified by table name t in COLUMNS, or TABLES consists of just the table name t.

- $blkjoin(e_1,\ e_2,\ \lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda u.if\ c = scan\pi_l(scanObj\ y \mid y \lhd u)\ then\ e_4\ else\ false,\ e_5,\ e_6) \rightsquigarrow blkjoin(e_1,\ e_2,\ \lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS and t.l = c, $\lambda u.e_4,\ e_5,\ e_6)$, if c is a constant, l is a column name qualified by table name t in COLUMNS, or TABLES consists of just the table name t.

- $idxjoin(e_1,\ e_2,\ e_3,\ \lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda u.if\ c = scan\pi_l(scanObj\ y \mid y \lhd u)\ then\ e_5\ else\ false,\ e_6,\ e_7,\ e_8) \rightsquigarrow idxjoin(e_1,\ e_2,\ e_3,\ \lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS and t.l = c, $\lambda u.e_5,\ e_6,\ e_7,\ e_8)$, if c is a constant, l is a column name qualified by table name t in COLUMNS, or TABLES consists of just the table name t.

- $idxjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda u.if\ c =$

$scan\pi_l(scanObj\ y \mid y \lhd u)\ then\ e_2\ else\ false,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8) \rightsquigarrow idxjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS and t.l = c, $\lambda u.e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8)$, if c is a constant, l is a column name qualified by table name t in COLUMNS, or TABLES consists of just the table name t.

- $prejoin(\lambda x.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda x.e_1$, $\lambda x.e_2) \rightsquigarrow prejoin(\lambda x.sqlserver$ select $t_1.l_1$, ..., $t_n.l_n$ from TABLES where CONDITIONS, $\lambda x.e_1,\ \lambda x.e_2)$, if the following three conditions are true. First, every occurrence of x in e_1 and e_2 is in a subexpression of the form $scan\pi_l(e \mid y \lhd x)$ or $putscan\pi_l(e \mid y \lhd x)$. Second, the l_i's are all the l's indicated in such subexpressions. Third, COLUMNS is either the wildcard * and TABLES has exactly one table t and each t_i is t; or $t_1.l_1$, ..., $t_n.l_n$ are strictly included in COLUMNS.

- $blkjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda y.e_2,\ e_3,\ e_4,\ \lambda y.e_5,\ \lambda y.e_6) \rightsquigarrow blkjoin(\lambda u.sqlserver$ select $t_1.l_1,\ ...,\ t_n.l_n$ from TABLES where CONDITIONS, $\lambda y.e_2,\ e_3,\ e_4,\ \lambda y.e_5,\ \lambda y.e_6)$, if the following three conditions are true. First, every occurrence of y in e_2, e_5, and e_6 is in a subexpression of the form $\pi_l\ y$, $scan\pi_l(e \mid w \lhd y)$, or $putscan\pi_l(e \mid w \lhd y)$. Second, the l_i's are all the l's indicated in such subexpressions. Third, COLUMNS is either the wildcard * and TABLES has exactly one table t and each t_i is t; or $t_1.l_1$, ..., $t_n.l_n$ are strictly included in COLUMNS.

- $blkjoin(e_1,\ e_2,\ \lambda v.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda z.e_4, \lambda y.\lambda z.e_5,\ \lambda y.\lambda z.e_6) \rightsquigarrow blkjoin(e_1,\ e_2,\ \lambda v.sqlserver$ select $t_1.l_1,\ ...,\ t_n.l_n$ from TABLES where CONDITIONS, $\lambda z.e_4, \lambda y.\lambda z.e_5,\ \lambda y.\lambda z.e_6)$, if the following three conditions are true. First, every occurrence of z in e_4, e_5, and e_6 is in a subexpression of the form $\pi_l\ z$, $scan\pi_l(e \mid w \lhd z)$, or $putscan\pi_l(e \mid w \lhd z)$. Second, the l_i's are all the l's indicated in such subexpressions. Third, COLUMNS is either the wildcard * and TABLES has exactly one table t and each t_i is t; or $t_1.l_1$, ..., $t_n.l_n$ are strictly included in COLUMNS.

- $idxjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $\lambda y.e_2,$ $\lambda y.e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.e_7,\ \lambda y.e_8) \rightsquigarrow idxjoin(\lambda u.sqlserver$ select $t_1.l_1,\ ...,\ t_n.l_n$ from TABLES where CONDITIONS, $\lambda y.e_2,\ \lambda y.e_3,\ e_4,\ e_5,\ e_6,\ \lambda y.e_7,\ \lambda y.e_8)$, if the following

three conditions are true. First, every occurrence of y in e_2, e_3, e_7, and e_8 is in a subexpression of the form $\pi_l\ y$, $scan\pi_l(e \mid w \lhd y)$, or $putscan\pi_l(e \mid w \lhd y)$. Second, the l_i's are all the l's indicated in such subexpressions. Third, COLUMNS is either the wildcard * and TABLES has exactly one table t and each t_i is t; or $t_1.l_1$, ..., $t_n.l_n$ are strictly included in COLUMNS.

- $idxjoin(e_1,\ e_2,\ e_3,\ \lambda v.sqlserver$ select COLUMNS where CONDITIONS, $\lambda z.e_5,\ \lambda z.e_6,\ \lambda y.\lambda z.e_7,\ \lambda y.\lambda z.e_8) \rightsquigarrow idxjoin(e_1,\ e_2,\ e_3,\ \lambda v.sqlserver$ select $t_1.l_1$, ..., $t_n.l_n$ from TABLES where CONDITIONS, $\lambda z.e_5,\ \lambda z.e_6,\ \lambda y.\lambda z.e_7,\ \lambda y.\lambda z.e_8)$, if the following three conditions are true. First, every occurrence of z in e_5, e_6, e_7, and e_8 is in a subexpression of the form $\pi_l\ z$, $scan\pi_l(e \mid w \lhd z)$, or $putscan\pi_l(e \mid w \lhd z)$. Second, the l_i's are all the l's indicated in such subexpressions. Third, COLUMNS is either the wildcard * and TABLES has exactly one table t and each t_i is t; or $t_1.l_1$, ..., $t_n.l_n$ are strictly included in COLUMNS.

- $idxjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $e_2,\ \lambda y.\pi_l\ y,\ \lambda v.sqlserver$ select COLUMNS' from TABLES' where CONDITIONS', $e_5,\ \lambda z.\pi_{l'}\ z,\ e_7,\ e_8) \rightsquigarrow prejoin(\lambda u.sqlserver$ select COLUMNS" from TABLE" where CONDITIONS" and t.l = t'.l', $\lambda x.if\ e_2(putObj$ (REFORMAT $(scanObj\ x)))\ then\ if\ e_5\ (putObj($ REFORMAT′ $(scanObj\ x)))\ then\ e_7($REFORMAT $(scanObj\ x))$ (REFORMAT′ $(scanObj\ x))\ else\ false\ else\ false,\ \lambda x.e_8($REFORMAT $(scanObj\ x))($REFORMAT′ $(scanObj\ x)))$, if the nine conditions below hold. First, COLUMNS is not the wildcard *. Second, COLUMNS' is not the wildcard *. Third, COLUMNS" is the union of COLUMNS and COLUMNS". Fourth, TABLES" is the union of TABLES and TABLES". Fifth, CONDITIONS" is the union of CONDITIONS and CONDITIONS". Sixth, t.l is in COLUMNS. Seventh, t'.l' is in COLUMNS'. Eighth, REFORMAT is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS is $t_1.l_1$, ..., $t_n.l_n$. Last, REFORMAT′ is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS' is $t_1.l_1$, ..., $t_n.l_n$.

- $idxjoin(\lambda u.sqlserver$ select COLUMNS from TABLES where CONDITIONS, $e_2,\ e_3,\ \lambda v.sqlserver$ select COLUMNS' from TABLES' where CONDITIONS', $e_5,\ e_6,\ \lambda y.\lambda z.if\ \pi_l\ y\ = \pi_{l'}\ z\ then\ e_7\ else\ false,\ e_8) \rightsquigarrow prejoin(\lambda u.sqlserver$ select COLUMNS" from

TABLES" where CONDITIONS" and t.l = t'.l', $\lambda x.if\ e_2(putObj$ (REFORMAT($scanObj$ x))) $then\ if\ e_5(putObj$(REFORMAT'($scanObj\ x$))) $then\ if\ e_3$(REFORMAT($scanObj\ x$)) $= e_6$(REFORMAT'($scanObj\ x$)) $then\ (\lambda y.\lambda z.e_7)$(REFORMAT($scanObj\ x$))(REFORMAT' $scanObj\ x$) $else\ false\ else\ false\ else\ false$, $\lambda x.e_8$(REFORMAT($scanObj\ x$))(REFORMAT' ($scanObj\ x$))), if the nine conditions below hold. First, COLUMNS is not the wildcard *. Second, COLUMNS' is not the wildcard *. Third, COLUMNS" is the union of COLUMNS and COLUMNS". Fourth, TABLES" is the union of TABLES and TABLES". Fifth, CONDITIONS" is the union of CONDITIONS and CONDITIONS". Sixth, t.l is in COLUMNS. Seventh, t'.l' is in COLUMNS'. Eighth, REFORMAT is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS is $t_1.l_1$, ..., $t_n.l_n$. Last, REFORMAT' is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS' is $t_1.l_1$, ..., $t_n.l_n$.

- *blkjoin* $(\lambda u.\ sqlserver$ select COLUMNS from TABLES where CONDITIONS, e_2, $\lambda v.sqlserver$ select COLUMNS' from TABLES' where CONDITIONS', e_4, $\lambda y.\lambda z.if\ \pi_l\ y = \pi_{l'}\ z\ then\ e_5\ else\ false$, $e_6) \rightsquigarrow prejoin(\lambda u.sqlserver$ select COLUMNS" from TABLES" where CONDITIONS" and t.l = t'.l', $\lambda x.if\ e_2(putObj$(REFORMAT($scanObj\ x$))) $then\ if\ e_4(putObj$(REFORMAT'($scanObj\ x$))) $then\ (\lambda y.\lambda z.e_5)$ (REFORMAT($scanObj\ x$)) (REFORMAT' ($scanObj\ x$)) $else\ false\ else\ false$, $\lambda x.e_6$(REFORMAT ($scanObj\ x$))(REFORMAT'($scanObj\ x$))), if the nine conditions below hold. First, COLUMNS is not the wildcard *. Second, COLUMNS' is not the wildcard *. Third, COLUMNS" is the union of COLUMNS and COLUMNS". Fourth, TABLES" is the union of TABLES and TABLES". Fifth, CONDITIONS" is the union of CONDITIONS and CONDITIONS". Sixth, t.l is in COLUMNS. Seventh, t'.l' is in COLUMNS'. Eighth, REFORMAT is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS is $t_1.l_1$, ..., $t_n.l_n$. Last, REFORMAT' is $\lambda y.(l_1 : \pi_{l_1}\ y,\ \ldots, l_n : \pi_{l_n}\ y)$ where COLUMNS' is $t_1.l_1$, ..., $t_n.l_n$.

Here is an example to illustrate these rules. Consider the query $putObj \bigcup\{\bigcup\{if\ \pi_A\ x = \pi_B\ y\ then\ \{(\pi_C\ x,\ \pi_D\ y)\}\ else\ \{\} \mid x \in scanObj(sqlserver$ select * from U where 1 = 1)$\} \mid y \in scanObj(sqlserver$ select * from V where 1 = 1)$\}$. It joins two tables U and V from the server. It is rewritten to $prejoin(\lambda u.sqlserver$ select U.C, V.D from U, V where U.A = V.B, $\lambda u.true$, $\lambda x.putSingletonSet(putPair(C : putscan\pi_C(y \mid y \lhd x), D : putscan\pi_D(y \mid y \lhd x))))$.

The join is migrated to the server. This rewritten query can itself be simplified to *sqlserver* select U.C, V.D from U, V where U.A = V.B, if a few additional rules are made available.

7.6 Pushing operations to ASN.1 servers

Relational servers are not the only kind of external data source. Suppose we have servers that provide data in ASN.1 [105] format. Data of this format are essentially hierarchically structured with records, variants, sets, and lists freely combined. While the data are richer in structure, these servers may have weaker capability than a relational server. For example, they may not be able to perform a join.

This section outlines how a simple ASN.1 server can be exploited. This simple server is comparable to the real server to be described in greater detail in Chapter 9 and Section 8.6. Without loss of generality, consider a server for a look-up table containing entries of type $string \times t$, where the first component is the look-up key and the second component is kept in ASN.1 format. Let this server be captured by the new construct in Figure 7.8. The

$$\frac{e_1 : string \quad e_2 : \{t\} \to \{s\}}{asnserver(e_1, e_2) : [\![\{s\}]\!]}$$

Figure 7.8: The construct for accessing an ASN.1 server.

semantics is as follow. The server looks up all entries with keys matching e_1. It applies e_2 to transform the second projection of these entries. The result is returned as a token stream. The server is not very powerful and it can only perform projection and flattening. Hence e_2 is required to be a simple sequence of projections and flattening. For simplicity here, I directly write this e_2 as PATH where PATH is either empty, denoting identity; is .I PATH' denoting relational projection on column I followed by PATH'; or is .E PATH' denoting flattening followed by PATH'.

Here is a short example to illustrate this construct. Assuming that the records on the server have type $string \times (seq : \{(giim : string, \ldots)\}, \ldots)$. Then the query $asnserver(\text{“}hemoglobin\text{”}, .\mathsf{seq.E.giim})$ finds all records matching hemoglobin and produces a flat relation containing all their giim components.

Some basic rules for moving operations to the ASN.1 server are given below.

- $putscanSet(e_1 \mid x \lhd asnserver(e_2, \mathsf{PATH})) \leadsto putscanSet(e_1[putPair(l : y)/x] \mid y \lhd asnserver(e_2, \mathsf{PATH.l}))$, if every occurrence of x in e_1 is in a subexpression of the form $scan\pi_l(e \mid w \lhd x)$ or $putscan\pi_l(e \mid w \lhd x)$.
- $putscanSet(putscanSet(e_1 \mid y \lhd x) \mid x \lhd asnserver(e_2, \mathsf{PATH})) \leadsto putscanSet(e_1 \mid y \lhd asnserver(e_2, \mathsf{PATH.E}))$, if x is not free in e_1.

As an example of the effect of these rules, consider the query $putObj\ \bigcup\{\bigcup\{\{\pi_{giim}\ x\} \mid x \in \pi_{seq}\ y\} \mid y \in scanObj(asnserver(\text{“}hemoglobin\text{”},\ ))\}$. In this query, the server returns all records about hemoglobin. The local system has to perform projections and flattening to obtained the desired output. It is rewritten to $asnserver(\text{“}hemoglobin\text{”}, \mathsf{seq.E.giim})$. The server is now made to return the desired output directly.

In the next chapter, I present the results of several simple experiments on the optimizations discussed in this and in the previous chapters.

Chapter 8

Potpourri of Experimental Results

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
> Donald Knuth

This chapter reports experiments I did on my prototype implementation. I divide the experiments into six groups, as outlined below. The measurements indicate that the optimizations suggested in Chapters 6 and 7 result in performance improvement.

All experiments were performed on a SPARC Server 690MP Model 51 with 128 megabytes of memory. The load on the machine was light when I ran my experiments. The other heavyweight that was running during some of my experiments was a 3-satisfiability checker. As my machine has two processors, the impact of this 3-satisfiability checker and other processes on my performance measurements was not significant.

I record only total time (the time taken from query submission to the printing of the last character of the reply), response time (the time taken from query submission to the printing of the first two characters of the reply), and peak memory usage. All timing data are system time as measured by ML's (the host language of my system) internal clock mechanism. All memory usage data are measured using Unix's top command. The peak memory usage data are approximate. This is because (1) ML does not collect all garbage immediately, (2)

ML may ask for a larger amount of memory than it needs, and (3) the operating system may hand ML more memory than it asks for. Nevertheless, the measurements are a good reflection of the general memory demand characteristics of various optimization options in my system.

Organization

Section 8.1. This group of experiments tests the pipelining rules of Chapter 6 in typical single-table scan situations.

Section 8.2. This group of experiments tests the effect of caching and indexing when some input databases are small enough to fit easily into main memory. This tests the rules suggested in Section 7.1.

Section 8.3. The third group tests the join optimization rules presented in Sections 7.2 and Section 7.3. I also measure the effectiveness of these rules against the rules for small relations.

Section 8.4. This group of experiments tests the effect of caching large intermediate results in joins on disk. These experiments essentially exercises those rules given in Section 7.4.

Section 8.5. This group of experiments tests the rules given in Section 7.5. These rules are for migrating selections, projections, and joins on external data imported from Sybase sources to their originating Sybase servers.

Section 8.6. This last group of experiments tests the rules presented in Section 7.6. These rules are for pushing projections and variant analysis on non-relational external data imported from ASN.1 [151] sources to their originating ASN.1 servers.

8.1 Loop fusions

The experiments in this section are concerned with the use of my pipelining rules. These are the rules described in Chapter 6. I annotate the data obtained when none of these rules are used by the tag **NoPipeline**. I annotate the data obtained when only input pipelining rules (that is, those given in Sections 6.1 and 6.2) are used by **InOnly**. I annotate the data obtained when only output pipelining (that is, those given in Sections 6.1 and 6.3) are used by **OutOnly**. I annotate the data obtained when all rules in Section 6.4 are used by **AllPipeline**.

The databases involved in this set of experiments are denoted DB1. They all have type `[(#1:[(#1:int, #2:int, #3:int)], #2:int)]`. All integers appearing in them are between 0 and 1000. All subcomponent lists in them contain between 1 and 100 small records. The size in terms of number of records of these databases ranges from 1000 large records to 10000 large records. The size in terms of number of bytes ranges from 1.37 megabytes to 14.1 megabytes. While these sizes are less than the size of the main memory on my test machine, they give a sense of how performance relates to database size.

This group of queries are all scans of a single table. They contain many opportunities for all forms of pipelining and they contain many opportunities for input filtering, and hence the output is considerably smaller in size than the input.

Experiment A

The Query

```
primitive egA == \DB1 => [z | (_,\z) <--- DB1];
```

The query above is a very simple relational projection on the second column of DB1. Recall that the second column of DB1 has type `int` while the first column has type `[(#1:int, #2:int, #3:int)]`. So this query allows a great opportunity for input filtering.

The measurements for this experiment are given in Figure 8.1. **InOnly** and **AllPipeline** perform significantly better than **OutOnly** and **NoPipeline** in all aspects. In terms of response time, **AllPipeline** is instantaneous because this query involves no search. **OutOnly**, **InOnly**, and **NoPipeline** have response times proportional to the size of DB1 because they cannot produce any output until the whole projection operation is completed. **InOnly** is faster than the other two because it does not load full input records into memory and so requires less time to complete the projection operation. In terms of memory demand, **AllPipeline** and **InOnly** are much better than the other two. This outcome is a direct consequence of the fact that **OutOnly** and **NoPipeline** do no input pipelining and must load the entire DB1, including its huge first column, into main memory.

		Size of DB1 in Megabytes					
		1.37	2.79	5.63	8.37	9.88	14.12
Total Time in Seconds	**AllPipeline**	65.75	134.33	275.73	402.8	485.09	718.79
	InOnly	63.92	133.43	274.36	400.16	482.95	713.5
	OutOnly	98.92	208.6	431.16	618.48	765.87	1097.97
	NoPipeline	97.9	209.32	450.45	614.22	753.05	1092.1
Response Time in Seconds	**AllPipeline**	0.07	0.13	0.07	0.09	0.07	0.07
	InOnly	53.15	106.92	226.4	319.64	393.12	578.24
	OutOnly	98.92	162.42	337.08	485.18	590.45	855.73
	NoPipeline	97.9	164.4	354.63	479.58	597.44	862.51
Peak Memory in Megabytes	**AllPipeline**	15	15	15	15	15	15
	InOnly	16	15	16	15	15	15
	OutOnly	23	39	62	85	106	140
	NoPipeline	23	39	61	86	99	134

Figure 8.1: The performance measurements for Experiment A.

Notice that the memory usage for the last column of **OutOnly** and **NoPipeline** slightly

exceeded the 128 megabytes of main memory on my machine. (This excessive amount of memory usage is not my programming fault — version 0.93 of the Standard ML of New Jersey is just not economical when it comes to using memory.) As a result, the inefficiency of **OutOnly** and **NoPipeline** can be attributed partially to increased paging activities. The impact of paging activities on the cost of **OutOnly** and **NoPipeline** is likely to be much more significant for a DB1 that is considerably larger than 14 megabytes, making **AllPipeline** even more desirable.

Experiment B

The Query

```
primitive egB == \DB1 => [XY | (\XY,\z) <--- DB1, 150 < z, z < 200];
```

This query contains a relational projection and a relational selection. Here, the first column of DB1 is projected. Since the first column of DB1 has type `[(#1:int,#2:int, #3:int)]`, the output is a list of lists. This query differs from the previous in two aspects: the output is larger and some search is required.

Performance Report

The measurements for this experiment are given in Figure 8.2. In terms of total time, **AllPipeline** and **InOnly** retain their performance advantage because they are able to discard much of the input before constructing the output. **AllPipeline** is better than **InOnly** here as it does not need to accumulate any output. In terms of response time, **AllPipeline** beats the other three. **OutOnly**, **NoPipeline**, and **InOnly** are slow because they need to process DB1 completely before printing. In terms of peak memory usage, **AllPipeline** peaks at about 15 megabytes. As before, **OutOnly** and **NoPipeline** is proportional to size of DB1. This time **InOnly** begins to need space to store its output, which is much larger than in the previous query.

		Size of DB1 in Megabytes					
		1.37	2.79	5.63	8.37	9.88	14.12
Total	**AllPipeline**	71.99	153.17	294.2	450.47	505.72	734.79
Time in	**InOnly**	72.2	149.76	309.11	462.9	523.95	763.7
Seconds	**OutOnly**	104.42	214.58	424.51	677.28	753.22	1105.06
	NoPipeline	103.32	219.55	420.92	650.84	730.74	1105.06
Response	**AllPipeline**	2.26	0.51	0.22	2.19	0.48	0.34
Time in	**InOnly**	55.87	115.43	230.0	338.87	392.84	567.6
Seconds	**OutOnly**	78.28	161.73	314.11	498.89	560.6	813.27
	NoPipeline	79.12	165.29	314.99	490.84	551.83	808.26
Peak	**AllPipeline**	15	15	15	15	15	15
Memory in	**InOnly**	15	15	16	18	18	20
Megabytes	**OutOnly**	24	39	64	92	101	135
	NoPipeline	25	40	65	92	103	142

Figure 8.2: The performance measurements for Experiment B.

Experiment C

The Query

```
primitive egC == \DB1 =>
  [(xy,z) | (\XY,\z) <--- DB1, 150 < z, z < 200,
     \xy & (#1:\x, ...) <--- XY, 150 < x, x < 200];
```

This query contains a relational unnesting and two selections. One of the selections is on a level-one attribute `z`. The other one is on a level-two attribute `x`. This query differs from the previous one in two aspects: (1) the previous query returns the first column of DB1 without looking at it while this query returns a selected portion of it, and thus (2) this query gives smaller output.

Performance Report

The performance data for this experiment is given in Figure 8.3 and is similar to that for the previous experiment: **AllPipeline** is best in all aspects, **InOnly** a close second, while **OutOnly** and **NoPipeline** remain close together at a distant joint-third position. The effect of the selection on `x` in this query is visible in the memory usage numbers. Here **InOnly** does not need as much space as in the previous query.

Experiment D

The Query

```
primitive egD == \DB1 =>
   [[(x,y) | (#1:\x,#2:\y,...) <--- XY] | (\XY,600) <--- DB1];
```

This query contains a selection and a projection. It differs from the earlier queries in two aspects. First, the projection is performed on each list in the first column of DB1; that

		Size of DB1 in Megabytes					
		1.37	2.79	5.63	8.37	9.88	14.12
Total Time in Seconds	**AllPipeline**	65.5	149.1	261.71	431.55	458.44	656.4
	InOnly	63.62	141.13	261.45	431.8	463.77	702.01
	OutOnly	91.01	217.28	405.97	656.9	709.72	1018.37
	NoPipeline	95.36	204.87	391.37	638.8	688.91	1056.22
Response Time in Seconds	**AllPipeline**	2.45	1.14	2.45	2.59	1.49	0.71
	InOnly	51.74	114.13	211.71	345.84	377.89	568.52
	OutOnly	74.09	168.34	314.25	504.68	557.33	785.74
	NoPipeline	74.33	161.09	304.66	506.16	535.19	824.64
Peak Memory in Megabytes	**AllPipeline**	15	15	15	15	15	15
	InOnly	15	15	15	15	15	15
	OutOnly	30	39	64	91	97	134
	NoPipeline	26	38	64	87	102	134

Figure 8.3: The performance measurements for Experiment C.

is, this operation is a nested projection. Second, the condition used in the selection is an equality test instead of a range condition. As this selection condition is harder to meet, more interesting response time behavior can be expected.

Performance Report

The measurements for this experiment are given in Figure 8.4. The relative performance of the optimizations remains the same: **AllPipeline** and **InOnly** are significantly better than **OutOnly** and **NoPipeline**. The most interesting change is in the response time of **AllPipeline**. Its fluctuations reflect the positions of the first record in the input that meets the strict selection condition. The other three do not fluctuate for the simple reason that they do not produce any output until everything else is completed and hence do not depend on when the first record meeting the condition is found. Nevertheless, it is clear that **AllPipeline** cannot respond slower than them.

Sample output of optimization for experiment D

I display below the various versions of query `egD` produced by my system. The output is taken straight off the optimizer and is in Kleisli's (my query system) internal format. While it is not easy to read, I think it is very educational to see the general changes made by the various optimization rules.

The output of **NoPipeline**

The numbers prefixed with a `v` are variable names. `StdIn` is Kleisli's standard scanning procedure. Notice that the output command `putObj` is placed at the outermost position while the input command `scanObj` is placed innermost. The positioning of these commands corresponds to the three distinct phases of input, execute, and print in query evaluation.

```
(putObj) @ (extList \v15=>(if ((#2) @ v15 = 600) then (etaList)
```

		Size of DB1 in Megabytes					
		1.37	2.79	5.63	8.37	9.88	14.12
Total Time in Seconds	**AllPipeline**	68.68	137.88	286.49	417.46	501.65	702.62
	InOnly	67.41	143.77	284.42	434.26	496.27	693.09
	OutOnly	97.47	215.37	405.36	671.25	737.36	1039.24
	NoPipeline	101.12	210.82	407.96	645.46	721.92	1036.63
Response Time in Seconds	**AllPipeline**	53.18	20.71	118.53	71.94	6.66	24.13
	InOnly	54.94	117.25	233.13	352.49	405.86	567.42
	OutOnly	77.31	169.19	318.0	522.84	579.12	804.24
	NoPipeline	80.37	170.81	323.76	508.42	563.11	819.78
Peak Memory in Megabytes	**AllPipeline**	15	15	15	15	15	15
	InOnly	15	15	15	15	16	15
	OutOnly	25	38	62	92	103	145
	NoPipeline	25	38	63	90	101	135

Figure 8.4: The performance measurements for Experiment D.

```
@ (extList \v16=>(etaList) @ (#1: (#1) @ v16, #2: (#2) @ v16)) @
(#1) @ v15 else [])) @ (scanObj) @ (StdIn) @ "1KDB1"
```

The output of **InOnly**

All operations such as `extList`, which work only on complex objects, have been replaced by corresponding operations such as `scanList` which works on token streams. Notice that the `scanObj` prevously placed next to `StdIn` has disappeared, as the query execution phase and the input phase are now merged.

```
(putObj) @ (scanList \v525=>(if ((scan#2 scanObj) @ v525 = 600)
then (scan#1 \v528=>(etaList) @ (scanList \v527=>(etaList) @
(#1: (scan#1 scanObj) @ v527, #2: (scan#2 scanObj) @ v527)) @
v528) @ v525 else [])) @ (StdIn) @ "1KDB1"
```

The output of **OutOnly**

All operations such as `extList` are replaced by corresponding token stream operations such as `putList`. The `putObj` in the original query has disappeared, as the output phase and the query execution phase are now merged.

```
(putList \v512=>(if ((#2) @ v512 = 600) then (putEtaList) @
(putList \v513=> (putEtaList) @ putrecord(#1: (putObj) @ (#1) @
v513, #2: (putObj) @ (#2) @ v513)) @ (#1) @ v512 else putEmptyList))
@ (scanObj) @ (StdIn) @ "1KDB1"
```

The output of **AllPipeline**

The two previous optimizations are combined here. In addition, places where a `scan` operation is immediately followed by a `put` operation are replaced by a `putscan` operation.

This eliminates the complex object that must be built for `scan` to communicate with `put`. Notice that both the `scanObj` and the `putObj` in the original query have disappeared, as all three phases of query process are now merged.

```
(putscanList \v47=>(if ((scan#2 scanObj) @ v47 = 600) then
(scan#1 \v51=> (putEtaList) @ (putscanList \v50=>(putEtaList) @
putrecord(#1: (scan#1 \v52 =>v52) @ v50, #2: (scan#2 \v53=>v53) @
v50)) @ v51) @ v47 else putEmptyList)) @ (StdIn) @ "1KDB1"
```

Notes

The performance characteristics for this group of queries are quite simple and can be summarized as follow.

In terms of total time, the performance is always linearly proportional to the size of the input table. **InOnly** and **AllPipelines** stay close together and are about 30% more efficient than **OutOnly** and **NoPipeline**. This improvement is expected because **InOnly** and **AllPipelines** pipeline and filter their inputs. Thus they assemble only a small portion of the input table into memory (and only a fragment of this small portion at a time), giving them an advantage over the other two, which fully assemble their inputs into memory before doing anything.

In terms of response time, the performance of **InOnly**, **OutOnly**, and **NoPipeline** depend linearly on the size of their inputs; but **AllPipeline** depends only on the position of the first record that meets the search conditions of the scan. The first three are much more sluggish than **AllPipeline**. This outcome is expected under a uniform distribution. Suppose every record has an equal chance of meeting the search condition. Then the probability of the first record meeting the search condition being near the end of the table table exponentially decreases with respect to its position. Hence, the first record meeting the search condition has a high probability of being near the front of the input, especially when the search condition is generous. **InOnly** is also observed to be about 30% faster in response time

than **OutOnly** and **NoPipeline** . This outcome is expected because it assembles its input into memory 30% quicker than the other two.

In terms of peak memory usage, the performance of **NoPipeline** and **OutOnly** depend linearly on the size of their inputs; **InOnly** and **AllPipeline** do not depend on the size of their inputs. This outcome is because **NoPipeline** and **OutOnly** load whole input tables into memory before doing anything but **InOnly** and **AllPipeline** load a fragment at a time. The size of output has little effect in the experiments of this section because it is rather small in comparison to the size of input.

The fact that total time improvement is linearly proportional to the size of input when all pipelinings are done is consistent with the observations in Chapter 6. However, recall that in Chapter 6 an improvement of approximately 50% is predicted. So there is a 20% improvement that is missing from my experimental numbers.

This discrepancy between theory and practice can be explained. The cost model used in Chapter 6 is overly simple. That cost model assumes that all basic operations have unit cost. This assumption is not correct. That cost model also assumes that the overhead of process suspension and resumption required in the implementation of laziness can be ignored. This assumption is also not correct. Furthermore, that cost model ignores the effect of operating system costs such as page faults. This assumption is also not correct.

In the design of my prototype, I have chosen simplicity over efficiency in several places. So this 20% difference can be reduced by replacing certain modules with more efficient ones. The modules for token streams are a good place to start. Token streams are the backbone in my system for laziness. Currently, they are implemented using an essentially linear representation. This implementation does allow us to skip over portions of a token stream quickly, but it does not allow us to update a token stream quickly. As a result, an operation such as *putUnionSet* has linear cost instead of constant cost.

8.2 Caching and indexing small relations

The experiments in this section deal with the special situation where some input databases are small enough to fit completely into main memory. These are the rules described in Section 7.1. In this set of experiments, I artificially set my rules up so that they consider anything below 1 megabyte small.

The baseline for this set of experiments uses all the pipelining rules but no caching rules. I annotate the baseline data with the tag **NoCache**. The data where only general caching rules are used are tagged with **CacheOnly**. The data where both general and indexed caching are used are tagged with **IndexCache**. For this set of experiments, I record only total times and response times. Peak memory requirements are omitted because they never exceed 15 megabytes.

There are two set of databases used in this set of experiments. The first set contains one database DB1 of type `{(#1:{(#1:int, #2:int, #3: int)}, #2: int}`. It has 1000 records and is 1.4 megabytes in size. All integers in DB1 are from 0 to 1000. The nested sets in the first column contain 1 to 100 records. The second set of databases are denoted DB2 and they have type `{(#1:int, #2:int)}`. Again the integers in DB2 are from 0 to 1000. The size in terms of number of records of DB2 ranges from 100 to 4000 records; in terms of bytes, from 1982 to 79129 bytes.

Experiment E

The Query

```
primitive egE == (\DB1,\DB2) =>
  { (x,v) | (\XY,\z) <- DB1,
            (z,\v) <- DB2, 200 < v, v < 250,
            (#1:\x,...) <- XY, 150 < x, x < 200 };
```

This query contains two selections, a join, and an unnesting. Note that DB2, the small relation, is the inner relation of the join. I am not using any form of join optimization in this group of experiments, so a naive nested-loop join algorithm is used. Thus DB2 has to be read for each record in DB1, the outer relation. So we have an opportunity to do caching and indexing.

Performance Report

Figure 8.5 gives the total time performance of **NoCache**, **CacheOnly**, and **IndexCache** as DB2 varies from 1000 to 4000 records. The improvement of **CacheOnly** and **IndexCache** over **NoCache** is dramatic. The savings of **CacheOnly** comes from loading DB2 only once; it still has to iterate over the whole of DB2 for each record in DB1. **IndexCache** also builds an index on the first column of DB2, so it does not need to iterate over the whole of DB2 for each record in DB1. Thus **IndexCache** is the most efficient of the three here.

Figure 8.5 also gives the response time performance for the same experiments. The response time of **IndexCache** follows a trend that is different from **CacheOnly** and **NoCache** because of an implementation decision. **CacheOnly** keeps the cached version of DB2 in token stream form and builds the cache token-by-token as the query is processed. So its response time mirrors that of **NoCache**. **IndexCache** first brings DB2 completely into main memory, builds the index, and only then begins the query. Hence it has a response time delay proportional to the size of DB2.

Experiment F

The Query

```
primitive egF == (\DB1,\DB2) =>
  { { (x, v, z) |
        (#1:\x, #2:\y, ...) <- XY ,
        (y, \v) <- DB2 , 300 < x, x < 500 } |
```

		Number of Records in DB2			
		1000	2000	3000	4000
Total Time in Seconds	**NoCache**	1369.67	2807.09	4311.69	5295.06
	CacheOnly	341.52	623.47	892.14	1137.29
	IndexCache	72.39	86.4	80.51	80.5
Response Time in Seconds	**NoCache**	27.51	22.74	14.3	11.82
	CacheOnly	9.96	9.54	7.34	7.49
	IndexCache	3.83	5.58	5.87	7.07

Figure 8.5: The performance measurements for Experiment E.

```
(\XY, \z) <- DB1, 300 < z, z < 400 };
```

This query is a nested query containing two selections, a projection, and a join. It differs from the previous query in that its join is a nested join. That is, each set in the first column of DB1 is joined with DB2. This query again offers good opportunity for caching.

Performance Report

Figure 8.6 gives the total time performance of **NoCache**, **CacheOnly**, and **IndexCache** when DB2 varies from 100 records to 3000 records. The performance is that both **IndexCache** and **CacheOnly** are better than **NoCache**. This improvement comes entirely from not loading DB2 repeatedly. (The impact of page faults can be ignored as no more than 15 megabytes of main memory are used in this experiment.) The difference between **IndexCache** and **CacheOnly** is very small here because the sets involved in the joins are all small. (Recall that all sets in the first column of DB1 have less than 100 elements.)

Figure 8.6 also gives the response time measurements for the same experiments. All three have the same response time trend. This is because the initial search is on the second column of DB1. The joins cannot be performed until a record is found and hence DB2 is

not touched until then. Hence response time does not depend on what is done to DB2.

		Number of Records in DB2				
		100	500	1000	2000	3000
Total	**NoCache**	190.57	729.63	1409.39	2787.36	4145.29
Time in	**CacheOnly**	101.53	213.83	355.49	643.35	924.35
Seconds	**IndexCache**	101.53	215.47	354.61	637.52	912.18
Response	**NoCache**	4.31	4.73	6.95	5.78	7.26
Time in	**CacheOnly**	4.15	4.68	5.65	5.33	7.63
Seconds	**IndexCache**	2.18	2.67	3.39	4.68	7.58

Figure 8.6: The performance measurements for Experiment F.

Experiment G

The Query

```
primitive egG == \DB2 =>
  {(x,w) | (\x,\y) <- DB2, (y,\z) <- DB2, (z,\w) <- DB2};
```

This query is a typical chain query. It has two joins. Moreover, both joins have equality tests as their join condition. The effect of indexing is expected to be very significant.

Performance Report

Figure 8.7 gives the total time measurements of **NoCache**, **CacheOnly**, **IndexCache** when DB2 varies from 100 records to 4000 records. As expected, **IndexCache** is many orders of magnitude more efficient than the other two. The improvement of **CacheOnly** over **NoCache** comes only from not loading the input table repeatedly. However, the improvement of **IndexCache** is greater because in addition to not loading the input table

repeatedly, its index allows it to use only a linear number of equality tests rather than a cubic number.

Figure 8.7 also gives the response time measurements of the same experiments. The response time of **CacheOnly** and **NoCache** are better than that of **IndexCache**. This outcome is because the latter must completely build the index on the first column of DB2 before everything else.

		Number of Records in DB2					
		100	500	1000	2000	3000	4000
Total	**NoCache**	12.9	483.17	2753.32	16273.16	-	-
Time in	**CacheOnly**	3.49	104.8	536.16	3459.29	-	-
Seconds	**IndexCache**	1.67	4.54	8.28	34.86	94.14	209.31
Response	**NoCache**	0.46	0.38	0.4	0.45	-	-
Time in	**CacheOnly**	0.51	0.56	0.48	0.56	-	-
Seconds	**IndexCache**	1.06	3.05	4.44	7.02	10.29	13.58

Figure 8.7: The performance measurements for Experiment G.

Sample output of optimization for experiment G

The output of **NoCache**

Only the effects of pipelining are present in the **NoCache** output of the query `egG`.

```
(putscanSet \v102=>(putscanSet \v105=>(if ((scan#1 scanObj)
@ v105 = (scan#2 scanObj) @ v102) then (putscanSet \v106=>(if
((scan#1 scanObj) @ v106 = (scan#2 scanObj) @ v105) then
(putEtaSet) @ putrecord(#1: (scan#1 \v107=>v107) @ v102,
#2: (scan#2 \v108=>v108) @ v106) else putEmptySet)) @ (StdIn) @
```

```
"4KDB2" else putEmptySet)) @ (StdIn) @ "4KDB2") @ (StdIn) @ "4KDB2"
```

The output of **CacheOnly**

Notice that all the `StdIn @ "4KDB2"` in the original query are now replaced by `Cache @ "4KDB2"`. `Cache` is the name of a new primitive that I have added to the system to implement caching of small input. Two things should be pointed out. (1) It is the optimizer that discovers that `Cache` can be profitably used in this query. (2) Even though `Cache @ "4KDB2"` appears three times in the transformed query, the file `4KDB2` is read only once. `Cache` keeps an internal record of files that have already been read.

```
(putscanSet \v134=>(putscanSet \v137=>(if ((scan#1 scanObj) @
v137 = (scan#2 scanObj) @ v134) then (putscanSet \v138=>(if
((scan#1 scanObj) @ v138 = (scan#2 scanObj) @ v137) then (putEtaSet)
@ putrecord(#1: (scan#1 \v139=>v139) @ v134, #2: (scan#2 \v140=>v140)
@ v138) else putEmptySet)) @ (Cache) @ "4KDB2" else putEmptySet))
@ (Cache) @ "4KDB2") @ (Cache) @ "4KDB2"
```

The output of **IndexCache**

Two occurrences of `Cache` in indexable positions have been replaced by `Index`. `Index` is the new primitive I have introduced to cache and index small input. It has three parameters: the name of the input file is in field `#1`, the indexing function to be used is in field `#3`, an integer for booking keeping purposes is in field `#2`. (This booking keeping field is really an identifier for the index to be used. Its value is computed automatically by **IndexCache**.) `Index` takes in these parameters and produces an indexing function. When this function is applied to a key, all the matching entries are returned in a set.

```
(putscanSet \v118=>(scan#2 \v127=>(putSet \v125=>(putSet
\v128=>(putEtaSet) @ putrecord(#1: (scan#1 \v123=>v123) @ v118,
```

```
#2: (putObj) @ (#2) @ v128)) @ ((Index) @ (#1: "4KDB2", #2: 1,
#3: #1)) @ (#2) @ v125) @ ((Index) @ (#1: "4KDB2", #2: 0, #3: #1))
@ (scanObj) @ v127) @ v118) @ (Cache) @ "4KDB2"
```

8.3 Joins

The **CacheOnly** and **IndexCache** optimizations have two weaknesses. The first is that they cannot be applied to relations that are too large to fit into memory. The second is that they always cache and index an entire relation even when only a portion of it is needed. The blocked nested-loop join and the indexed blocked-nested-loop join, described respectively in Section 7.2 and Section 7.3 are generalization of these two optimizations that do not have the two defficiencies stated above.

This set of experiments uses two sets of databases, DB1 and DB2. DB1 has type `{(#1: [(#1: int, #2: int, #3:int)], #2: int)}`. Its size in terms of number of records ranges from 1000 records to 10000 records and in terms of bytes from 0.6 megabytes to 7.2 megabytes; so a typical record is about 700 bytes in size. The lists in its first column have lengths between 1 and 50 records. DB2 has type `{(#1: int, #2: int)}`. Its size in terms of number of records ranges from 1000 records to 4000 records and in terms of bytes from 19 kilobytes to 79 kilobytes; so a typical record is about 20 bytes in size. All integers in these databases are between 0 and 1000.

I use the tag **BlockOnly** to denote data obtained using only the blocked nested-loop join optimization rules described in Section 7.2. I use the tag **IndexBlock** to tag data obtained using the indexed blocked-nested-loop join optimization rules described in Section 7.3. The **IndexCache** optimization of the Section 7.1 is used as a baseline for comparison.

Experiment H

The Query

```
primitive egH == (\DB1,\DB2) =>
  { (x,v) | (\XY,\z) <- DB1, 300 < z, z < 800,
            (z,\v) <- DB2, 200 < v, v < 250,
            (#1:\x,...) <--- XY, 150 < x, x < 200 };
```

This query is a variation of experiment E. The purpose is to see how well these two optimizations are doing relative to the **IndexCache** optimization of the previous section. Recall that **IndexCache** builds indices on small relations. So in this query, **IndexCache** indexes on DB2. On the other hand, **IndexBlock** builds indices on outer relations of joins. So in this query, **IndexBlock** indexes on DB1. The performance report below has to be interpreted with accordingly.

Performance Report (Blocking factor at 1000 records. DB2 at 1000 records. DB1 varying)

These measurements, in Figure 8.8, are obtained by fixing the blocking factor at 1000 records, DB2 at 1000 records, and varying DB1 from 1000 records to 10000 records.

The total time performance of all three optimizations is linearly proportional to size of DB1 because all three have chosen DB1 to be the outer relation for the join. **BlockOnly** and **IndexBlock** is less efficient than **IndexCache** for this range of input data because **IndexCache** reads DB2 once only, while the other two has to read it as many times as there are blocks in DB1. **BlockOnly** is worse than **IndexBlock** because the latter exploits the equality test in the join condition to use indexed access.

The response time performance of all three optimizations is stable because the main search condition is on DB1, the outer relation. **BlockOnly** and **IndexBlock** respond quickly when DB1 has only 1000 records. The reason is that many of these records do not satisfy the predicate `300 < z < 800` on the DB1 and hence the block buffer for **BlockOnly** and

		Number of Records in DB1			
		1000	4000	7000	10000
Total	**IndexCache**	39.32	144.8	243.76	342.21
Time in	**BlockOnly**	55.35	189.26	327.91	455.54
Seconds	**IndexBlock**	42.1	166.78	295.84	417.98
Response	**IndexCache**	8.18	4.71	4.5	6.74
Time in	**BlockOnly**	44.8	75.26	78.28	77.8
Seconds	**IndexBlock**	33.06	64.64	68.29	66.48
Peak	**IndexCache**	30	15	15	15
Memory in	**BlockOnly**	17	26	27	27
Megabytes	**IndexBlock**	17	20	20	20

Figure 8.8: The performance measurements for Experiment H with DB1 varying.

IndexBlock are only partially filled when DB1 is fully read. As a consequence, the buffer is released for the block join earlier. **IndexCache** turns in a lower response time when DB1 is at 1000 records for a reason revealed in the memory usage data.

The memory usage of **IndexCache** is high when DB1 is at 1000 records. This is because DB1 at this size is below 1 megabyte, the threshold for a relation to be considered small by **IndexCache**. So **IndexCache** goes ahead and caches DB1, as well as DB2. Fortunately, the cache is kept in token stream form, so this only delays the response time of **IndexCache** by several seconds. In contrast, the memory usage for **BlockJoin** and **IndexBlock** is lower when DB1 is small because there are not enough records to fill the block.

Performance Report (Blocking factor at 1000 records. DB1 at 7000 records. DB2 varying)

The data in Figure 8.9 is obtained by fixing the blocking factor at 1000 records, DB1 at 7000 records, and letting DB2 to vary from 1000 records to 4000 records. The response time behavior of these optimizations is unremarkable; the table is omitted.

		Number of Records in DB2			
		1000	2000	3000	4000
Peak	**IndexCache**	15	16	17	17
Memory in	**BlockOnly**	27	27	27	27
Megabytes	**IndexBlock**	20	20	20	20
Total	**IndexCache**	243.76	254.39	256.99	262.61
Time in	**BlockOnly**	327.91	376.73	396.24	424.74
Seconds	**IndexBlock**	295.84	313.34	320.35	329.36

Figure 8.9: The performance measurements for Experiment H with DB2 varying.

The total time performance of these three optimizations with respect to size of DB2 is expected to show the following trends: **BlockOnly** is proportional to the product of the blocking factor and the size of DB2, **IndexBlock** is proportional to the product of the log of the blocking factor and the size of DB2, and **IndexCache** is proportional to the size of DB2. The data is consistent with these expectations.

The peak memory usage pattern of all three optimizations is stable. **IndexCache** is the least costly in this aspect because it stores only DB2, which is a small relation. The memory requirement for **BlockOnly** and **IndexBlock** is dictated by the blocking factor. However, **BlockOnly** has to keep buffer blocks for both the outer relation (DB1) and the inner relation (DB2). So it uses more memory than **IndexBlock** which keeps buffer blocks for the outer relation only.

Performance Report (DB1 at 7000 records. DB2 at 4000 records. Blocking factor varying)

The measurements in Figure 8.10 are obtained by fixing DB1 at 7000 records, DB2 at 4000 records, and varying the blocking factor from 500 records to 3000 records.

The response time and memory usage pattern are directly affected by the blocking factor in

		Blocking Factor (Number of Records)			
		500	1000	2000	3000
Peak Memory in Megabytes	**BlockOnly**	20	27	37	37
	IndexBlock	17	20	27	34
Total Time in Seconds	**BlockOnly**	435.01	424.74	408.53	414.1
	IndexBlock	339.33	329.36	321.58	324.46
Response Time in Seconds	**BlockOnly**	50.95	85.91	148.27	220.97
	IndexBlock	35.17	70.51	137.31	205.71

Figure 8.10: The performance measurements for Experiment H with blocking factor varying.

the expected manner — the larger the blocking factor, the slower the response and the more memory used. In addition, **BlockOnly** needs more memory than **IndexBlock** because it caches both inner and outer blocks while the latter caches only the outer block.

The total time performance is more complex. Total time performance steadily improves as blocking factor increases until it reaches 2000 records and then begins to deteriorate. A larger blocking factor leads to less blocks and larger blocks. Less blocks means DB2 is read a smaller number of times; this saves time for both **BlockOnly** and **IndexBlock**. On the other hand, larger blocks are more costly to assemble. If DB2 is small, than loading it a few more times may not be that costly. I conjecture that the optimum blocking factor for this experiment must be about 2000 records. In any case, **IndexBlock** is observed to be better than **BlockOnly**.

Sample output of optimization for experiment H

The output of **IndexCache**

The baseline for experiment H is **IndexCache**. As can be seen, all pipelinings have been done and DB2 has also been identified for runtime indexing.

```
(putscanSet \v152=>(if ((scan#2 scanObj) @ v152 < 800) then (if
(300 < (scan#2 scanObj) @ v152) then (scan#2 \v161=>(putSet \v159=>
(if ((#2) @ v159 < 250) then (if (200 < (#2) @ v159) then (scan#1
putscanLS \v156=>(if ((scan#1 scanObj) @ v156 < 200) then (if (150
< (scan#1 scanObj) @ v156) then putEtaSet @ putrecord(#1: (scan#1
\v157=>v157) @ v156, #2: (putObj) @ (#2) @ v159) else putEmptySet)
else putEmptySet)) @ v152 else putEmptySet) else putEmptySet)) @
((Index) @ (#1: "4KDB2", #2: 0, #3: #1)) @ (scanObj) @ v161) @ v152
else putEmptySet) else putEmptySet)) @ (StdIn) @ "4KDB1"
```

The output of **BlockOnly**

The join present in `egH` is identified by **BlockOnly**, which rewrites the query to use the `BlkJoin` operator. `BlkJoin` is a new function I injected into the system to implement blocked nested-loop joins. It has six input parameters: the generator of the outer relation is in field `#1`, the selection predicate on the outer relation is in field `#2`, the generator for the inner relation is in field `#3`, the selection predicate on the inner relation is in field `#4`, the join predicate is in field `#5`, and the transformation to be performed on the join is in field `#6`.

```
(BlkJoin) @ (#1: \v300=>(StdIn) @ "4KDB1", #2: \v293=>(if (300 <
(scan#2 scanObj) @ v293) then ((scan#2 scanObj) @ v293 < 800) else
false), #3: \v302=>(StdIn) @ "4KDB2", #4: \v313=>(if (200 < (scan#2
scanObj) @ v313) then ((scan#2 scanObj) @ v313 < 250) else false),
#5: \v307=>\v306=>((#1) @ v306 = (#2) @ v307), #6: \v307=>\v306=>
(putLS \v308=>(if ((#1) @ v308 < 200) then (if (150 < (#1) @ v308)
then (putEtaSet) @ putrecord(#1: (putObj) @ (#1) @ v308, #2: (putObj)
@ (#2) @ v306) else putEmptySet) else putEmptySet)) @ (#1) @ v307)
```

The output of **IndexBlock**

The join condition produced by **BlockOnly** is `\v307=>\v306=>((#1) @ v306 = (#2) @ v307)`, which is an equality test saying that the second column of the outer relation has to equal the first column of the inner relation. Thus an index can be created on the second column on the outer relation and the first column of the inner relation can be used as the probe for the indexed blocked-nested-loop join. **IndexBlock** makes this discovery and replaces `BlkJoin` with `IdxJoin`. `IdxJoin` is the function I have inserted into the system to implement the indexed blocked-nested-loop join. It has eight input parameters: the generator for the outer relation is in field `#1`, the selection predicate on the outer relation is in field `#2`, the function for extracting the key to be used for indexing the outer relation is in field `#3`, the generator for the inner relation is in field `#4`, the selection predicate on the inner relation is in field `#5`, the function for extracting the probe value from the inner relation is in field `#6`, the join predicate is in field `#7`, and the transformation to be performed on the join is in field `#8`.

```
(IdxJoin) @ (#1: \v273=>(StdIn) @ "4KDB1", #2: \v266=>(if (300 <
(scan#2 scanObj) @ v266) then ((scan#2 scanObj) @ v266 < 800) else
false), #3: #2, #4: \v275=> StdIn @ "4KDB2", #5: \v286=>(if (200 <
(scan#2 scanObj) @ v286) then ((scan#2 scanObj) @ v286 < 250) else
false), #6: #1, #7: \v280=>\v279=>true, #8: \v280 => \v279=>(putLS
\v281=>(if ((#1) @ v281 < 200) then (if (150 < (#1) @ v281) then
(putEtaSet) @ putrecord(#1: (putObj) @ (#1) @ v281, #2: (putObj) @
(#2) @ v279) else putEmptySet) else putEmptySet)) @ (#1) @ v280)
```

Notes

I have only implemented these two join operators for sets. Bags should also benefit from similar operators and optimizations. However, they cannot be applied to lists because ordering of elements in a list must be respected.

8.4 Caching inner relations

The plan for the basic Kleisli system has an important restriction: it is not allowed to use any disk space during query evaluation. All the optimizations I have described so far respect this restriction. This restriction has an undesirable consequence for joins. Recall that the inner relation in a join has to be read as many times as there are blocks in the outer relation. These repetitious reads are not a problem if the inner relation is a base relation existing on disk. However, if the inner relation is actually a subquery, then this subquery must be recomputed as many times as there are blocks in the outer relation. The recomputation of the subquery can be expensive.

In order to avoid recomputation of the subquery, its result must be stored. As the result can be potentially large, to be safe, it has to be written to a disk. Section 7.4 relaxes the no-disk restriction and introduces a new set of optimization rules to take advantage of the disk by caching large intermediate results to it. This section contains an experiment to illustrate the effects of these new rules on the total time and the response time performance of the system. Data obtained when caching is turned on is annotated by the suffix **Cache**. The blocking factor used in this experiment is 3000 records.

Three set of databases are used. The first set, DB1, contains only one database of type `{(#1: [(#1:int, #2: int, #3:int)], #2: int)}`. This database has 4000 records and occupies 2.8 megabytes; so a typical record is about 700 bytes in size. The lists in it are all between 0 and 50 records. All integers in it are between 0 and 1000. The second set, DB2, has only one database of type `{(#1: int, #2: int)}`. This database has 4000 records and occupies 79 kilobytes. Its integers are all between 0 and 1000. The third set, DB3, has four databases of type `{(#1: int, #2: int)}`. They contain 7000 records, 10000 records, 15000 records, and 20000 records respectively. In terms of bytes, they range from 150 kilobytes to 219 kilobytes. All integers in these databases are in the range –2000 to 2000.

Experiment I

The Query

```
primitive egI == (\DB3, \DB2, \DB1) =>
  { (z, u, {v | \v & (#3: \w, ...) <--- XY, z < w, w < u})  |
      (_ ,\z) <- DB3, z < 600, (z,\u) <- DB2,
      (\XY,u) <- DB1, 300 < u, u < 500};
```

This query involves a join of three relations plus a nested selection and returns a nested relation. Since this is a three-way join, one of the binary join has to be the inner loop and is a potentially expensive subquery to be repeated many times.

Performance Report

Figure 8.11 gives the total time measurements when the number of records in DB3 grows from 7000 to 20000. The cached versions of the blocked nested-loop join and the indexed blocked-nested-loop join are significantly better than the corresponding uncached versions. The numbers also indicate that indexing leads to much faster queries.

Figure 8.11 also gives the response time measurements for the same experiment. It is found that **IndexBlockCache** has a slower response time than **IndexBlock**. On the other hand, **BlockOnlyCache** has a significantly faster response time than **BlockOnly**. In `egI`, as can be seen in the optimizer outputs given later, the subquery is the join between DB2 and DB3. This subquery is the inner relation used in both **BlockOnly** and **IndexBlock**. The blocked nested-loop join algorithm loads both its inner and outer relations block-by-block, using a blocking factor of 3000 records in this experiment. Hence the first 3000 records (in the output of this subquery) that satisfy the inner predicate `300 < u < 500` must be produced before we can proceed further, if the blocked nested-loop join algorithm is used. On the other hand, the indexed blocked-nested-loop join algorithm does not load its inner relation block-by-block. Hence we can proceed further without fully generating

the first 3000 records in the output of this subquery, if the indexed blocked-nested-loop join algorithm is used. This difference is the main reason for **IndexBlockCache** to respond slower than **IndexCache** and for **BlockOnlyCache** to respond faster than **BlockOnly**.

		Number of Records in DB3			
		7000	10000	15000	20000
Total Time in Seconds	**IndexBlockCache**	349.42	371.63	455.04	511.94
	IndexBlock	411.67	453.86	579.1	680.42
	BlockOnlyCache	2845.94	4579.39	6898.35	8119.74
	BlockOnly	5809.93	9481.0	13658.38	18396.81
Response Time in Seconds	**IndexBlockCache**	156.18	161.38	173.45	182.87
	IndexBlock	131.62	131.49	135.07	129.47
	BlockOnlyCache	1845.63	3432.57	5266.26	6249.69
	BlockOnly	2411.12	4288.0	6140.27	7893.52

Figure 8.11: The performance measurements for Experiment I.

Sample output of optimization for experiment I

The output of **BlockOnly**

This is the output of **BlockOnly** for query `egI`. As can be seen, two applications of `BlkJoin` is used to implement the three-way join.

```
(BlkJoin) @ (#1: \v327=>(StdIn) @ "4KDB1", #2: \v328=>true, #3:
\v338=>(BlkJoin) @ (#1: \v323=>(StdIn) @ "7KDB3", #2: \v315 =>
((scan#2 scanObj) @ v315 < 600), #3: \v325 => (StdIn) @ "4KDB2",
#4: \v344=>true, #5: \v332=>\v331=>((#1) @ v331 = (#2) @ v332),
#6: \v336=>\v337=>(putEtaSet) @ putrecord(#1: (putObj) @ v336,
#2: putObj @ v337)), #4: \v350=>(if (300 < (scan#2 scan#2 scanObj)
```

```
@ v350) then ((scan#2 scan#2 scanObj) @ v350 < 500) else false),
#5: \v340=>\v339=>((#2) @ v340 = (#2) @ (#2) @ v339), #6: \v340=>
\v339=> putEtaSet @ putrecord(#1: (putObj) @ (#2) @ (#1) @ v339,
#2: (putObj) @ (#2) @ (#2) @ v339, #3: (putLS \v349=>(if ((#2) @
(#1) @ v339 < (#3) @ v349) then (if ((#3) @ v349 < (#2) @ (#2) @
v339) then (putEtaSet) @ (putObj) @ v349 else putEmptySet) else
putEmptySet)) @ (#1) @ v340))
```

The output of **BlockOnlyCache**

There are two occurrence of the `BigCache` operator in the output of **BlockOnlyCache**. This operator takes in two parameters: a generator for the data to be cached is in field `#2` and an integer for housekeeping purposes is in field `#1`. (This housekeeping integer is generated automatically by **BlockOnlyCache**.) These two occurrences of `BigCache` cache the inner subqueries of the two respective `BlkJoin` produced by **BlockOnly**.

```
BlkJoin @ (#1: \v201=>(StdIn) @ "4KDB1", #2: \v202=>true, #3:
\v233 =>BigCache @ (#1: 0, #2: \v232=>(BlkJoin) @ (#1: \v197 =>
StdIn @ "7KDB3", #2: \v189=> ((scan#2 scanObj) @ v189 < 600), #3:
\v259 => BigCache @ (#1: 1, #2: \v258=> PreJoin @ (#1: \v261 =>
StdIn @ "4KDB2", #2: \v257=>(300 < (scan#2 scanObj) @ v257), #3:
putEtaSet)), #4: \v260=>true, #5: \v243=>\v242=>(if ((#1) @ v242
= (#2) @ v243) then ((#2) @ v242 < 500) else false), #6: \v237=>
\v238=> putEtaSet @ putrecord(#1: putObj @ v237, #2: putObj @ v238
))), #4: \v234=>true, #5: \v214=>\v213=>((#2) @ v214 = (#2) @ (#2)
@ v213), #6: \v214=>\v213 => putEtaSet @ putrecord(#1: putObj @
(#2) @ (#1) @ v213, #2: (putObj) @ (#2) @ (#2) @ v213, #3: (putLS
\v223=>(if ((#2) @ (#1) @ v213 < (#3) @ v223) then (if ((#3) @ v223
< (#2) @ (#2) @ v213) then (putEtaSet) @ (putObj) @ v223 else
putEmptySet) else putEmptySet)) @ (#1) @ v214))
```

The output of **IndexBlock**

Since the join conditions of both the joins identified by `BlkJoin` involve equality test, **IndexBlock** turns them into `IdxJoin`.

```
(IdxJoin) @ (#1: \v152=>(StdIn) @ "4KDB1", #2: \v153=>true, #3: #2,
#4: \v163=> (IdxJoin) @ (#1: \v148=>(StdIn) @ "7KDB3", #2: \v140 =>
((scan#2 scanObj) @ v140 < 600), #3: #2, #4: \v150=>StdIn @ "4KDB2",
#5: \v169 =>true, #6: #1, #7: \v157=> \v156=>true, #8: \v161=> \v162
=> (putEtaSet) @ putrecord(#1: putObj @ v161, #2: (putObj) @ v162)),
#5: \v175=>(if (300 < (scan#2 scan#2 scanObj) @ v175) then ((scan#2
scan#2 scanObj) @ v175 < 500) else false), #6: \v164=>(#2) @ (#2) @
v164, #7: \v165 => \v164 =>true, #8: \v165 => \v164 => putEtaSet @
putrecord(#1: putObj @ (#2) @ (#1) @ v164, #2: putObj @ (#2) @ (#2)
@ v164, #3: (putLS \v174=>(if ((#2) @ (#1) @ v164 < (#3) @ v174)
then (if ((#3) @ v174 < (#2) @ (#2) @ v164) then putEtaSet @ putObj
@ v174 else putEmptySet) else putEmptySet)) @ (#1) @ v165))
```

The output of **IndexBlockCache**

The two inner subqueries of the two `IdxJoin` are then cached by **IndexBlockCache** which inserted two `BigCache` operations into the query.

```
(IdxJoin) @ (#1: \v61=>(StdIn) @ "4KDB1", #2: \v62=>true, #3: #2,
#4: \v93=> BigCache @ (#1: 0, #2: \v92=> IdxJoin @ (#1: \v57 =>
StdIn @ "7KDB3", #2: \v49=>((scan#2 scanObj) @ v49 < 600), #3: #2,
#4: \v113=> BigCache @ (#1: 1, #2: \v112 => PreJoin @ (#1: \v115=>
(StdIn) @ "4KDB2", #2: \v111=>(if (300 < (scan#2 scanObj) @ v111)
then ((scan#2 scanObj) @ v111 < 500) else false), #3: putEtaSet)),
#5: \v114=>true, #6: #1, #7: \v97=> \v98=>true, #8: \v97=> \v98=>
```

```
(putEtaSet) @ putrecord(#1: (putObj) @ v97, #2: (putObj) @ v98))),
#5: \v94=> true, #6: \v73=>(#2) @ (#2) @ v73, #7: \v74=>\v73=>true,
#8: \v74=> \v73=> putEtaSet @ putrecord(#1: putObj @ (#2) @ (#1) @
v73, #2: (putObj) @ (#2) @ (#2) @ v73, #3: (putLS \v83=>(if ((#2) @
(#1) @ v73 < (#3) @ v83) then (if ((#3) @ v83 < (#2) @ (#2) @ v73)
then putEtaSet @ putObj @ v83 else putEmptySet) else putEmptySet))
@ (#1) @ v74))
```

8.5 Pushing operations to relational servers

Kleisli is an open system that allows new primitives, new optimization rules, new cost functions, new scanners, and new writers to be dynamically added. This allows me to connect it to many external databases. Many of these databases are Sybase relational databases. Suppose a query involves some of these databases. It is generally more efficient to move as many operations to these databases as possible than to try to bring the data in and to process them locally within Kleisli. I have implemented the optimization rules of Section 7.5 to migrate projections, selections, and joins on external Sybase data to their source database systems.

In the experiment below, the suffix **Sel** indicates use of selection pushing, the suffix **Sel-Proj** indicates use of both selection pushing and projection pushing, and the suffix **Sel-ProjJoin** indicates use of all three relational optimizations. **Sel** is turned on throughout the experiment. Three large tables on a real Sybase database are used in this experiment. Their sizes are 40739, 39120, and 40224 records for `object_genbank_eref`, `locus`, and `locus_cyto_location` respectively. The blocking factor used throughout the experiment is 20000 records.

Experiment K

The Query

```
primitive obj_gdb_eref == GDB @
  "select * from object_genbank_eref where 1 = 1";

primitive locus == GDB @ "select * from locus where 1 = 1";

primitive locus_cyto_loc == GDB @
  "select * from locus_cyto_location where 1=1";

primitive egK ==
  {(#genbank_ref: a, #locus_symbol: b, #loc_cyto_chrom_num: "22",
    #loc_cyto_band_start: d, #loc_cyto_band_end: e,
    #loc_cyto_band_start_sort: f, #loc_cyto_band_end_sort: g) |
       (#genbank_ref:\a, #object_id:\h,
        #object_class_key:1, ...) <- obj_gdb_eref,
       (#locus_id:h, #loc_cyto_chrom_num:"22",
        #loc_cyto_band_start:\d, #loc_cyto_band_end:\e,
        #loc_cyto_band_start_sort:\f,
        #loc_cyto_band_end_sort:\g,...) <- locus_cyto_loc,
       (#locus_id:h, #locus_symbol:\b,...) <- locus };
```

This query contains two joins plus a number of seletions and projections. It has three subqueries `obj_gdb_eref`, `locus`, and `locus_cyto_loc` that bring in three remote relations from GDB, a genome database curated by the Welch Medical Library of Johns Hopkins. Notice that `egK`, the query we want to execute, itself is SQL-free. I left the SQL syntax in the three subqueries for illustration purposes; they can be avoided as well.

Performance Report

The performance of **IndexBlockCacheSelProjJoin** is the best in every aspect. This outcome is to be expected because it manages to push the entire query to the Sybase server. The performance of **IndexBlockCacheSel** is the worse in every aspect. This poor performance is because **IndexBlockCacheSel** does not push projections to Sybase and hence has to deal with full records every time, in contrast to the other three which are transmitted only the relevant fields of records. See Figure 8.12.

The amount of data in this experiment is sufficiently large to show the difference in the performance of **IndexBlock** and **IndexBlockCache**. From the table below, **IndexBlockCache** takes about 4 seconds, estimated from the difference in response time when both **Sel** and **Proj** are performed, to write the cache file. However, the cache shaves more than 20 seconds off the total time.

	Total Time (s)	Response Time (s)	Peak Memory (MB)
IndexBlockCacheSel	602.5	270.45	58
IndexBlockSelProj	146.6	62.69	30
IndexBlockCacheSelProj	118.68	66.83	30
IndexBlockCacheSelProjJoin	36.55	34.49	15

Figure 8.12: The performance measurements for Experiment K.

Sample output of optimization for experiment K

The output of **IndexBlockCache**

For the purpose of comparison, here is the output when no relational optimization is used. The two joins present in `egK` have been identified. Notice that the three SQL subqueries

appear in their original form.

```
(IdxJoin) @ (#1: \v234=>(Sybase) @ (#password: "bogus", #query:
"select * from locus where 1 = 1", #server: "WELCHSQL", #user:
"cbil"), #2: \v235=>true, #3: #locus_id, #4: \v262=>(BigCache)
@ (#1: 0, #2:\v261=> IdxJoin @ (#1:\v230=> Sybase @ (#password:
"bogus", #query: "select * from object_genbank_eref where 1 = 1",
#server: "WELCHSQL", #user: "cbil"), #2: \v223=>(
(scan#object_class_key scanObj) @ v223 = 1), #3: #object_id, #4:
\v272=> BigCache @ (#1: 1, #2: \v271=>(PreJoin) @ (#1: \v274 =>
Sybase @ (#password: "bogus", #query: "select * from
locus_cyto_location where 1 = 1", #server: "WELCHSQL", #user:
"cbil"), #2: \v270=>((scan#loc_cyto_chrom_num scanObj) @ v270 =
"22"), #3: putEtaSet)), #5: \v273=>true, #6: #locus_id, #7: \v239
=>\v238=>true, #8:\v244=>\v245=> putEtaSet @ putrecord(#1:putObj
@ v244, #2: (putObj) @ v245))), #5: \v263=>true, #6: \v247=>
(#object_id) @ (#1) @ v247, #7: \v248=>\v247=>true, #8: \v248=>
\v247=>(putEtaSet) @ putrecord(#genbank_ref: putObj @ #genbank_ref
@ #1 @ v247, #loc_cyto_band_end: putObj @ (#loc_cyto_band_end) @
(#2) @ v247, #loc_cyto_band_end_sort: (putObj) @
(#loc_cyto_band_end_sort) @ (#2) @ v247, #loc_cyto_band_start:
(putO bj) @ (#loc_cyto_band_start) @ (#2) @ v247,
#loc_cyto_band_start_sort: (putObj) @ (#loc_cyto_band_start_sort)
@ (#2) @ v247, #loc_cyto_chrom_num: (putObj) @ "22", #locus_symbol:
(putObj) @ (#locus_symbol) @ v248))
```

The output of **IndexBlockCacheSel**

Sel is turned on. Two selections `locus_cyto_chrom_num = "22"` and `object_class_key = 1` are moved into the SQL subqueries as a result.

```
(IdxJoin) @ (#1: \v140=>(Sybase) @ (#password: "bogus", #query:
"select * from locus where 1 = 1", #server: "WELCHSQL", #user:
"cbil"), #2: \v141=>true, #3: #locus_id, #4: \v168=>(BigCache) @
(#1: 0, #2: \v167=>(IdxJoin) @ (#1: \v136=> (Sybase) @ (#password:
"bogus", #query: "select * from object_genbank_eref where 1 = 1
and object_genbank_eref.object_class_key = 1", #server:"WELCHSQL",
#user: "cbil"), #2: \v129=>true, #3: #object_id, #4:\v138=> Sybase
@ (#password: "bogus", #query: "select * from locus_cyto_location
where 1 = 1 and locus_cyto_location.loc_cyto_chrom_num = '22'",
#server:"WELCHSQL", #user:"cbil"), #5:\v158=> true, #6: #locus_id,
#7: \v145=>\v144=>true, #8: \v150=> \v151=>(putEtaSet) @ putrecord
(#1: (putObj) @ v150, #2: (putObj) @ v151))), #5: \v169=>true, #6:
\v153=> (#object_id) @ (#1) @ v153, #7:\v154=>\v153=>true, #8:
\v154=>\v153=>(putEtaSet) @ putrecord(#genbank_ref: (putObj) @
#genbank_ref @ (#1) @ v153, #loc_cyto_band_end: (putObj) @
(#loc_cyto_band_end) @ (#2) @ v153, #loc_cyto_band_end_sort: putObj
@ (#loc_cyto_band_end_sort) @ (#2) @ v153, #loc_cyto_band_start:
(putObj) @ (#loc_cyto_band_start) @ (#2) @ v153,
#loc_cyto_band_start_sort: (putObj) @ (#loc_cyto_band_start_sort)
@ (#2) @ v153, #loc_cyto_chrom_num: (putObj) @ "22", #locus_symbol:
(putObj) @ (#locus_symbol) @ v154))
```

The output of **IndexBlockCacheSelProj**

Both **Sel** and **Proj** are turned on. The effect is that all projections are moved successfully to Sybase. This can be seen from the disappearance of the SQL wildcard * from the SQL subqueries.

```
(IdxJoin) @ (#1: \v88=>(Sybase) @ (#password: "bogus", #query:
"select locus.locus_id, locus.locus_symbol from locus where 1 = 1",
```

```
#server:"WELCHSQL", #user: "cbil"), #2: \v89=>true, #3: #locus_id,
#4: \v116=>(BigCache) @ (#1: 0, #2: \v115=>(IdxJoin) @ (#1: \v84=>
(Sybase) @ (#password: "bogus", #query: "select
object_genbank_eref.genbank_ref, object_genbank_eref.object_id
from object_genbank_eref where 1 = 1 and
object_genbank_eref.object_class_key = 1", #server: "WELCHSQL",
#user: "cbil"), #2: \v77=>true, #3: #object_id, #4: \v86=> Sybase
@ (#password: "bogus", #query: "select
locus_cyto_location.loc_cyto_band_end,
locus_cyto_location.loc_cyto_band_end_sort,
locus_cyto_location.loc_cyto_band_start,
locus_cyto_location.loc_cyto_band_start_sort,
locus_cyto_location.loc_cyto_chrom_num,
locus_cyto_location.locus_id from locus_cyto_location where 1 = 1
and locus_cyto_location.loc_cyto_chrom_num = '22'", #server:
"WELCHSQL", #user: "cbil"), #5: \v106=>true, #6: #locus_id, #7:
\v93=>\v92=>true, #8: \v98=>\v99=>(putEtaSet) @ putrecord(#1:
(putObj) @ v98, #2: (putObj) @ v99))), #5: \v117=>true, #6: \v101=>
(#object_id) @ (#1) @ v101, #7: \v102=>\v101=>true, #8: \v102=>
\v101=> putEtaSet @ putrecord(#genbank_ref: putObj @ (#genbank_ref)
@ (#1) @ v101, #loc_cyto_band_end: (putObj) @ (#loc_cyto_band_end)
@ (#2) @ v101, #loc_cyto_band_end_sort: (putObj) @
(#loc_cyto_band_end_sort) @ (#2) @ v101, #loc_cyto_band_start:
(putObj) @ (#loc_cyto_band_start) @ (#2) @ v101,
#loc_cyto_band_start_sort: (putObj) @ (#loc_cyto_band_start_sort) @
(#2) @ v101, #loc_cyto_chrom_num: (putObj) @ "22", #locus_symbol:
(putObj) @ (#locus_symbol) @ v102))
```

The output of **IndexBlockCacheSelProjJoin**

Sel, **Proj**, and **Join** are all turned on. The difference between this output and the previous ones is very significant. In particular, the two occurrences of `IdxJoin` have been pushed to Sybase.

```
(PreJoin) @ (#1: \v26=>(Sybase) @ (#password: "bogus", #query:
"select locus.locus_symbol, locus_cyto_location.loc_cyto_band_end,
locus_cyto_location.loc_cyto_band_end_sort,
locus_cyto_location.loc_cyto_band_start,
locus_cyto_location.loc_cyto_band_start_sort,
object_genbank_eref.genbank_ref from locus, locus_cyto_location,
object_genbank_eref where 1 = 1 and 1 = 1 and
locus_cyto_location.loc_cyto_chrom_num = '22' and 1 = 1 and
object_genbank_eref.object_class_key = 1 and
object_genbank_eref.object_id = locus_cyto_location.locus_id
and locus.locus_id = object_genbank_eref.object_id", #server:
"WELCHSQL", #user: "cbil"), #2: \v61=>true, #3: \v61=>(putEtaSet)
@ putrecord(#genbank_ref: (scan#genbank_ref \v64=>v64) @ v61,
#loc_cyto_band_end: (scan#loc_cyto_band_end \v65=>v65) @ v61,
#loc_cyto_band_end_sort: (scan#loc_cyto_band_end_sort \v66=>v66)
@ v61, #loc_cyto_band_start: (scan#loc_cyto_band_start \v67=>v67)
@ v61, #loc_cyto_band_start_sort: (scan#loc_cyto_band_start_sort
\v68=>v68) @ v61, #loc_cyto_chrom_num: (putObj) @ "22",
#locus_symbol: (scan#locus_symbol \v69=>v69) @ v61))
```

NOTES

I have only implemented these rules to deal with SQL expressions of the special form select COLUMNS from TABLES where CONDITIONS, as described in Section 7.5. It should not

be overly difficult to deal with SQL expressions that are of a more complicated form. A more challenging improvement is to attempt to shift the computation of certain aggregate functions, such as taking the average of a column, to the Sybase server as well.

8.6 Pushing operations to ASN.1 servers

The National Center for Biotechnology Information distributes their genetic database on a CD-ROM. This database is over 1.5 gigabytes in size and the schema for the MEDLINE portion of the database alone requires over 50 kilobytes to describe. This database is accessed from my system via a special C program asncpl. The C program is used as follows: asncpl -d DATABASE -s SELECTION -p PATH. The SELECTION is some boolean combination of keywords. Asncpl looks up all citations containing keywords satisfying the SELECTION. The PATH specifies the part of a citation to be returned. See Section 9.3 for more detail.

My system contains 8 optimization rules, two of which are described in Section 7.6, that push field projections and case analysis from our system down to asncpl by moving them into the PATH parameter. This section contains an experiment to demonstrate the effectiveness of these rules. I tag the results obtained using CD-ROM optimization rules by **WithFilter** and the results obtained without them by **NoFilter**.

Experiment J

The Query

Citations in this database comes from different sources. So they carry different but equivalent identifiers. This query takes in a keyword, finds citations containing that keyword, and returns the `giim` and `embl` identifiers of these citations. (Here `ASN` is the primitive corresponding to asncpl.)

```
primitive egJ == \keyword => {(id, acc)
```

```
| <#seq: (#id:\seq, ...)> <- ASN @ keyword
, <#giim : (#id:\id, ...)> <- seq
, <#embl : (#accession:\acc, ...)> <- seq} ;
```

Performance Report

The query is tested by supplying it with several randomly chosen keywords. They match from 1 citation to 294 citations. Citations can differ quite wildly in size and structure. This non-uniformity in the data is the main reason that the measurements are not very smooth. But it is clear that the performance of the system with these CD-ROM optimizations is significantly better than without the rules: total time is within seconds as opposed to minutes, response time is more stable. See Figure 8.13.

	Number of Citations Matched by Keywords							
	SMALL-SCALE (1)	NOSE (7)	EAR (12)	HEMO-GLOBIN (94)	EYE (115)	MOUTH (180)	PLASMA (292)	GLOBIN (294)
Total Time in Seconds								
NoFilter	2.34	3.04	4.06	52.67	28.82	37.07	98.07	138.33
WithFilter	2.58	2.62	2.58	3.98	3.85	4.52	6.48	7.12
Response Time in Seconds								
NoFilter	2.3	2.95	1.86	3.04	7.31	2.0	4.22	2.74
WithFilter	2.56	2.6	2.47	2.63	3.02	2.45	2.75	2.7

Figure 8.13: The performance measurements for Experiment J.

Sample output of optimization for experiment J

The output for experiment J, when the query `egJ` is applied to the keyword `hemoglobin`, is given below.

The output of **NoFilter**

The function `ASN` in the original query is defined in terms of a lower level primitive `Entrez` which directly interfaces with the C program `asncpl`. `Entrez` has three parameters. The name for the NCBI database to be accessed is in field `#db`; the nucleic acid database (`na`) is used. The selection condition is in field `#select`; set to `hemeglobin` here. The filter path is in field `#path`; it is set to the root `Seq-entry` by default.

```
(putscanSet scanCase #seq: \v499=>(scan#id putscanSet scanCase
#giim: \v506 => (scan#id putscanSet scanCase #embl: \v509 =>
(putEtaSet) @ putrecord(#1:(scan#id \v510=>v510) @ v506, #2:
(scan#accession \v511=>v511) @ v509) otherwise putEmptySet) @
v499 otherwise putEmptySet) @ v499 otherwise putEmptySet) @
Entrez @ (#db: "na", #path: "Seq-entry", #select: "hemoglobin")
```

The output of **WithFilter**

The optimized query produced by **WithFilter** contains the significant difference: the path parameter of `Entrez` is set to `Seq-entry.seq.id`. Thus it succeeds in pushing the selection on the variant tag `seq` and the projection on the field `id` in `egJ` to `asncpl`.

```
(putscanSet \v440=>(putscanSet scanCase #giim: \v459=>(putscanSet
scanCase #embl: \v464=>(putEtaSet) @ putrecord(#1: (scan#id \v477
=>v477) @ v459, #2: (scan#accession \v482=>v482) @ v464) otherwise
putEmptySet) @ v440 otherwise putEmptySet) @ v440) @ (Entrez) @
```

```
(#db: "na", #path: "Seq-entry.seq.id", #select: "hemoglobin")
```

8.7 Remarks

The ideas described in Chapters 6 and 7 are well-known principles for optimizing queries [6, 19, 75, 109, 148, 68]. They have well-understood characteristics. My experimental results in this chapter are consistent with the characteristics of these optimization ideas. Therefore, this chapter has provided some evidence that I have implemented the optimization rules described in Chapters 6 and 7 correctly.

I would like to point out that the optimizations tested in this chapter were implemented by me in less than three weeks. The rapid realization of these rules was made possible by the rule-based optimizer of Kleisli. More details of Kleisli can be found in Chapter 9. In particular, the actual programs that implement some of the optimization rules used in these experiments can be found in Section 9.4.

Chapter 9

An Open Query System in ML called Kleisli

Several researchers at the University of Pennsylvania Human Genome Center for Chromosome 22 are regularly required to write programs to query biological data. The task of writing these programs is taxing for two reasons. First, the information needed often resides in several data sources of very different nature; some are relational databases, some are structured text files, and others include output from special application programs. There is currently no high-level tool for combining data across such a diverse spectrum of sources. This lack of high-level tool makes it difficult to write programs that implement the queries because the programmer is forced to use many different application programming interfaces and programming languages such as SQL embedded in C. Second, the programmer must often resort to storing intermediate results for subsequent processing because the available tools are not flexible enough to retrieve the data into a desired form, which may not be relational.

Recall the query from Section 1.5: Find annotation information on the known DNA sequences on human chromosome 22, as well as information on sequences that are similar to them. Answering this query requires access to three different data sources — GDB[160],

SORTEZ[87], and Entrez[150]. GDB is a Sybase relational database located at Johns Hopkins and it contains marker information on chromosome 22. Entrez is a non-relational data source which contains several biological databases as well sequence similarity links. SORTEZ is our local relational database that we use to reconcile the difference between GDB identifiers and Entrez identifiers. To produce the correct groupings for this query, the answer has to be printed as a nested relation. Writing programs to execute queries such as this one is possible in C and SQL, but it would require an extraordinary amount of effort, and sharing common code between programs would be difficult.

I have built an open query system Kleisli and have implemented the collection programming language CPL, as a high-level query language for it. (The system is named after the mathematician H. Kleisli who discovered a natural transformation between monads [118]. As seen in Chapter 2, this transformation plays a central role in the manipulation of sets, bags, and lists in our system.) The openness of Kleisli allows the easy introduction of new primitives, optimization rules, cost functions, data scanners, and data writers. Furthermore, queries that need to freely combine external data from different sources are readily expressed in CPL. I claim that Kleisli, together with CPL, is a suitable tool for implementing the kind of queries on biological data sources that frequently need to be written. This chapter concentrates on connecting Kleisli and CPL to these systems.

Organization

Section 9.1. A description of the application programming interface of Kleisli is given. A short description of the compiler interface of Kleisli is given. A short description of how to use Kleisli and its data exchange format is given.

Section 9.2. An extended example is presented to illustrate programming with Kleisli's application programming interface. The example is the implementation of the indexed blocked-nested-loop operator described in Section 7.3.

Section 9.3. Examples are presented to illustrate the compiler interface of Kleisli. Specif-

ically, I show how a driver for Sybase servers, a driver for ASN.1 servers, and a sequence similarity package are introduced into Kleisli and CPL.

Section 9.4. Examples are presented to illustrate the rule-based optimizer that comes with Kleisli's compiler interface. Specifically, I show how to describe one of the indexed join rules in Section 7.3 and one of the Sybase rules in Section 7.5 to Kleisli.

Section 9.5. Two examples of genetic queries are presented to illustrate the use of Kleisli and CPL as a query interface for heterogenous biological data sources. One of these examples is actually a template for solving several real queries that were previously thought to be hard [57].

9.1 Overview of Kleisli

Kleisli is a prototype query system constructed on top of the functional programming language ML [144]. It is divided into two parts: the application programming interface and the compiler interface. The design and implementation of Kleisli emphasizes openness: new primitives, optimization rules, cost functions, data scanners, and data writers can all be dynamically introduced into the system. This openness, as shown in later sections, makes it possible to quickly extend Kleisli into a query interface for heterogenous biological data sources. This section presents an overview of the system.

Application programming interface

Kleisli supports sets, bags, lists, records, variants, token streams, and functions. These data types can be freely mixed and thus giving rise to a data model that is considerably richer and more flexible than the relational model. Each of these data types is encapsulated within the application programming interface by a collection of ML modules. The core of the collection-type modules (that is, those for sets, lists, bags, and token streams) are inspired principally by the work presented in earlier chapters.

An ML programmer can directly manipulate Kleisli complex objects via function calls to these modules. Each module consists of: a collection of canonical operators for that particular data type encapsulated by that module, additional operators designed for efficiency, additional operators that are frequently used composites of other operators, and conversion operators between ML and that Kleisli data type.

The modules for the Kleisli record type are worth a special mention. Kleisli supports record polymorphism [153, 165]. Its field-selection operator on records therefore has to be a function that can be applied to any record regardless of what fields it has, provided the field selected is present. (Record polymorphism is particularly important for accessing external data sources. It makes possible writing a program to select a field from an external table without knowing in advance what other columns are present in that table.) In the past, such an operator could not be implemented efficiently because the structure of the record is not known in advance. However, this Kleisli operator has efficient constant time performance. It is implemented using a technique developed recently by Remy [166].

The modules for Kleisli token stream are important as they provide Kleisli the mechanisms for laziness, data pipelining, and fast response. In an ordinary byte stream, such as ML's instream [11], a programmer must explicitly take care of bytes that have been read because reading is destructive. In contrast, a token stream is like a pure list and the programmer is free from such care. However, a token stream is also different from a list. A list has no internal structure; for example, after seeing an open bracket, it is not possible to skip directly to the matching close bracket. A token stream has internal structure and such direct jumps are supported.

An example is given in Section 9.2 to illustrate programming in ML with Kleisli's application programming interface.

Compiler interface

For ad hoc queries, it is frequently more productive to use a high-level query language. Kleisli has a compiler interface that supports rapid construction of high-level query languages. The interface contains modules which provide support for compiler and interpreter construction activities. This interface includes: (1) A general polymorphic type system that supports parametric record polymorphism [153, 165], and a type unification routine central to general type inference algorithms. (2) An abstract syntax structure for expressing Kleisli programs. Syntactic matching modulo renaming on abstract syntax objects and many other forms of manipulations are supported. Type inference on abstract syntax objects is also provided. (3) A rule-based optimizer and rewrite rule management. New optimization rules and cost functions can be registered dynamically. (4) External function management. New primitives can be programmed in ML and injected into the system dynamically. (5) Data scanner and writer management. New routines for scanning and writing external databases in various formats can be readily added to the system.

The importance of this interface is its flexibility and extensibility. A query language built on top of Kleisli can be readily customized for special application areas by injecting into the system relevant operators of the application area, rules for exploiting them in query optimization, appropriate cost estimation functions, and relevant scanners and writers.

Some of these extensible features are demonstrated in Section 9.3 and Section 9.4.

Using Kleisli

The general strategy for using Kleisli to query external databases is as follow. Special purpose programs, called data drivers, are used to manage low-level access to external databases and to return results as a text stream in a standard exchange format. (Any different exchange format can be used by any data driver, as long as a corresponding tokenizing program is provided.) The query specification for these programs, usually the arguments of the driver, is understood by Kleisli via a registration procedure. When a

query is executed by one of these drivers, the output stream is parsed on-the-fly and placed into a structured token stream, the universal data structure used by Kleisli for remote data access. At this point, the data has become an object that can be directly manipulated by Kleisli.

The components of this process are shown in Figure 9.1, which also reveals the presence of CPL. CPL is an example of a high-level query language rapidly constructed from the compiler interface of Kleisli. See Chapter 5 for an informal specification of CPL.

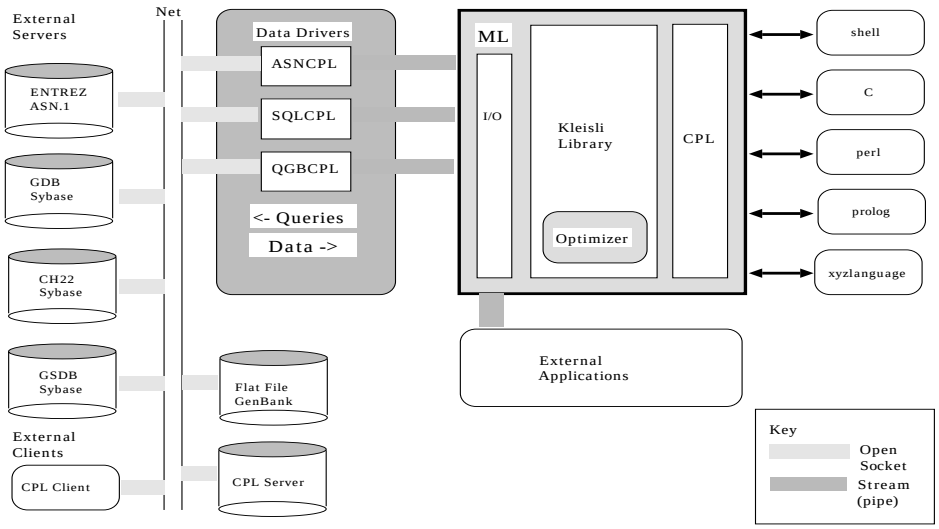


Figure 9.1: Using Kleisli to query external data sources.

The basic data exchange format of Kleisli can be described using the following grammar:

$$
\begin{array}{lcll}
V & ::= & C & \text{Integers, etc.} \\
 & | & \{V, \ldots, V\} & \text{Sets} \\
 & | & \{|V, \ldots, V|\} & \text{Bags} \\
 & | & [V, \ldots, V] & \text{Lists} \\
 & | & (L : V, \ldots, L : V) & \text{Records} \\
 & | & \langle L : V \rangle & \text{Variants}
\end{array}
$$

where L are record labels or variant labels. Functions and token streams cannot be transmitted. Punctuations such as commas and colons are optional. Using this basic format, an

external relational server can transmit a relation to Kleisli by laying it out according to the grammar like so:

```
{(#locus_id "YESP" #genbank_ref "D00333")
    .
    .
 (#locus_id "IGLV" #genbank_ref "D01059")}
```

Since relations from flat relational databases have regular structure, this format wastes bandwidth. So the standard exchange format of Kleisli makes special provision for it. Specifically, such a relation can be transmitted by first sending a header consisting of a sequence of labels prefixed by a dollar sign, then followed by records in the relation. The fields of each record is laid out according to the sequencing of labels in the header. In each record, instead of writing out their labels in full, a dollar sign is used. The {-bracket and the }-bracket enclosing the relation should each be prefixed by a dollar sign. The (-bracket and the)-bracket enclosing each record in the relation should each be prefixed by a dollar sign. For example, the same relation above can be transmitted as:

```
${ $#locus_id $#genbank_ref
   $( $ "YESP" $ "D00333" $),
     .
     .
   $( $ "IGLV" $ "D01059" $) $}
```

9.2 Programming in Kleisli

This section is an extended example using Kleisli's application programming interface to implement in ML the indexed blocked-nested-loop join operator described in Section 7.3. I present it in a bottom-up manner. Many low-level details of the Standard ML of New Jersey [11] and of the join algorithm appear in this example. Hence I explicitly point out

in various places where routines from Kleisli's application programming interface are used. Hopefully, this makes it easier to see the help provided by this interface.

The lowest level is the index structure itself. The Kleisli module `CompObjDict` is used. This module is a general module for managing adaptive trees [174] whose keys and nodes are Kleisli complex objects. Since distinct objects in an index can have the same key, the nodes are set to be Kleisli lists so that objects having the same keys are kept in the same list. Since the insertion routine `CompObjDict.insert` does not know that lists are used in this special situation, a wrapper routine has to be written as below. `CompoObjDict.peek` checks if a key is already in the index. The Kleisli token stream scan routine `Scan.ScanObj` is for converting a token stream object into a complex object; it behave more or less like the *scanObj* construct used in Chapter 6. The Kleisli list routine `CompObjList.Insert` is for list insertion. The Kleisli list routine `CompObjList.Eta` is for singleton list formation.

```
(* Dict is an adaptive tree managed by CompObjDict.
 * S is a token stream representing a complex object to be inserted.
 * I is the function for extracting the key of S.
*)
fun UpdateIndex(Dict, S, I) =
  let val Item = Scan.ScanObj S
      val Key  = I Item
  in (case CompObjDict.peek(Dict, Key)
      of SOME CO
      => CompObjDict.insert(Dict,Key,CompObjList.Insert(Item,CO))
      | NONE
      => CompObjDict.insert(Dict, Key, CompObjList.Eta Item))
  end
```

Recall from Section 7.3 that the indexed blocked-nested-loop join algorithm loads the outer relation block-by-block and builds an index for each block on-the-fly. Below is the ML function that loads one block and creates an index for it. The Kleisli token stream rou-

tine `TokenStream.GetToken` is for inspecting what the first token is. The Kleisli boolean complex object routine `CompObjBool.IfThenElse` is for doing conditional test. The Kleisli token stream routine `TokenStream.SkipObject` is for skipping over an entire object on the token stream. If the object has already been partially read, this routine has a cost proportional to the remaining portion of the object. So if the object has been fully read, this routine jumps straight to its end.

```
(* S      is a token stream representing blocks to be loaded.
 * P      is a predicate for deciding if a record is to be loaded.
 * I      is the indexing function.
 * !Limit is the blocking factor to be used.
 * Dict   is the index being created.
 * !SRef  is a token stream representing blocks remaining.
 * N      is the number of remaining slots in the index.
*)
fun LoadBlock'(SRef, P, I, Dict, 0)               = SOME Dict
|   LoadBlock'(ref NONE, P, I, Dict, N)           = NONE
|   LoadBlock'(SRef as ref(SOME S), P, I, Dict, N) =
      (case TokenStream.GetToken S
       of TokenStream.CloseSet => (SRef := NONE; SOME Dict)
       |  _ => let val (Q, Dict) = CompObjBool.IfThenElse(
                                   P S,
                                   fn() => (1, UpdateIndex(Dict, S, I)),
                                   fn() => (0, Dict))
                   val _ = SRef := SOME(TokenStream.SkipObject S)
               in LoadBlock'(SRef, P, I, Dict, N - Q) end)

fun LoadBlock(S, P, I) =
  let val SRef = ref(SOME(TokenStream.SkipToken S))
  in  fn() => LoadBlock'(SRef, P, I, (CompObjDict.mkDict()), !Limit)
```

```
end
```

Now the routine to loop over the outer relation and joins it with the inner relation has to be written. This routine is a ML function having three nested loops as follow. For each iteration of the outer loop, it loads and indexes one block from the outer relation using `LoadBlock`. Having built the index for this block, it proceeds to the middle loop, which is an iteration over the inner relation using `PutScanSet`. For each iteration of the middle loop, one record from the inner relation is loaded. If it satisfies the inner predicate `PredI`, then its key is computed using `Idx`. This key is used to probe the current index `Dict`. The inner loop is an iteration using `PutList2Set` over the list returned by the index probe. For each iteration of the inner loop, the join predicate `PredIO` is applied to check if current inner record and outer record qualify for the join. The transformation `Loop` is applied if they qualify. Several routines from the application programming interface are used. The Kleisli token stream copy routine `PutScan.CopySentinelSet` copies a set from a token stream, omitting the enclosing set brackets. The Kleisli token stream print routine `Put.PutList2Set` converts a Kleisli list to a set on a token stream. The Kleisli token stream print routine `PutScan.PutEmptySet` produces a token stream representing the empty set.

```
(* Outer  is function for loading next block of the outer relation.
 * Inner  is generator of the inner relation.
 * PredI  is filter on inner relation.
 * IdxI   is function for computing the probe value.
 * PredIO is join predicate.
 * Loop   is transformer of records to be joined.
 * State  is housekeeping data for token stream routines.
 * Cont   is continuation data for token stream routines.
 * TS     is token stream to pick up on completion of loop.
*)
fun LoopI(Outer,Inner,PredI,IdxI,PredIO,Loop,State,Cont) TS () =
  (case Outer()
   of SOME Dict => PutScan.CopySentinelSet
```

```
        (LoopI(Outer,Inner,PredI,IdxI,PredIO,Loop,State,Cont))
        State
        (PutScan.PutScanSet (fn X =>
           CompObjBool.IfThenElse(
             PredI X,
             fn() => let val CX = Scan.ScanObj X
                     in case CompObjDict.peek(Dict, IdxI CX)
                        of SOME CO => Put.PutList2Set(fn Y =>
                                        if PredIO Y CX
                                        then (Loop Y CX)
                                        else PutScan.PutEmptySet) CO
                         | NONE => PutScan.PutEmptySet end,
             fn() => PutScan.PutEmptySet))
           (Inner()))
        ()
    | NONE => Cont TS ())
```

Lastly, a ML function to take care of data conversion between ML and Kleisli has to be written. In this function, calls of the form SOMETHING.Km are for conversion from Kleisli to ML and calls of the form SOMETHING.Mk are for conversion from ML to Kleisli.

```
fun IdxJoinCode(X) =
  let
  val [Outer,PredO,IdxO,Inner,PredI,IdxI,PredIO,Loop]
           = CompObjRecord.KmTuple X
  val Inner = fn() => CompObjTokenStream.Km(
                CompObjFunction.Apply(Inner,CompObjUnit.Mk))
  val Outer = CompObjTokenStream.Km(
                CompObjFunction.Apply(Outer,CompObjUnit.Mk))
  val PredO = (CompObjFunction.Km PredO) o CompObjTokenStream.Mk
  val PredI = (CompObjFunction.Km PredI) o CompObjTokenStream.Mk
```

```
  val Idx0  = CompObjFunction.Km Idx0
  val IdxI  = CompObjFunction.Km IdxI
  val PredI0  = fn X => fn Y => CompObjBool.Km(CompObjFunction.Km(
                  (CompObjFunction.Km PredI0) X) Y)
  val Loop    = fn X => fn Y => CompObjTokenStream.Km(
                  CompObjFunction.Km((CompObjFunction.Km Loop) X) Y)
  val State   = PutScan.MkState()
  in
  CompObjTokenStream.Mk(
    PutScan.MkOpenSet
      (LoopI(LoadBlock(Outer, Pred0, Idx0),
            Inner, PredI, IdxI, PredI0, Loop,
            State, PutScan.MkCloseSet State))
      State
      TokenStream.NoMoreToken
      ())
  end
```

At this point, `IdxJoinCode` can be used within Kleisli as an operator for indexed blocked-nested-loop join. In order to use it within CPL, the high-level query language of Kleisli, it has to be registered. The registration is done using a simple function call to the compiler interface of Kleisli as below; see also Section 5.4. After that, `IdxJoin` can be used anywhere within CPL as a first-class citizen.

```
val IdxJoin = DataDict.RegisterCompObj(
  "IdxJoin",                      (* Name of primitive  *)
  CompObjFunction.Mk IdxJoinCode, (* Code of primitive  *)
  TypeInput.ReadFromString(       (* Type of primitive: *)
    "(#1:unit->[|{''1}|],"      ^ (*              Outer *)
    " #2:[|''1|]->bool,"        ^ (*              Pred0 *)
    " #3:''1->''2,"             ^ (*              Idx0  *)
```

```
" #4:unit->[|{''3}|],"      ^ (*              Inner *)
" #5:[|''3|]->bool,"        ^ (*              PredI *)
" #6:''3->''2,"             ^ (*              IdxI  *)
" #7: ''1 -> ''3 -> bool,"  ^ (*     join condition *)
" #8:''1->''3->[|{''4}|])"  ^ (*              Loop  *)
"-> [|{''4}|]"))
```

9.3 Connecting Kleisli to external data sources

As mentioned earlier, the compiler interface of Kleisli emphasizes openness. As a result, new scanners, writers, primitives, cost functions, and optimization rules are readily added to the system. This section concerns the use of this interface in connecting Kleisli to external data sources and in expanding Kleisli's collection of primitives.

Access to external systems are introduced into Kleisli and CPL using a three-step procedure. In the first step, a low-level access program or a data driver for the external system in question is written. If the program is not written in ML, the host programming language of Kleisli, it needs to be turned into a function in ML. In the second step, this function is registered as a scanner with Kleisli. In the third and final step, the scanner is turned into a Kleisli abstract syntax object and inserted into CPL as a full-fledged primitive. This three-step procedure is illustrated on some data drivers useful for querying biomedical data sources.

Querying relational databases

I have been given a general program for accessing Sybase relational database systems. This program,

sybcpl USER PASSWORD SERVER QUERY

is written in C [113]. It takes four parameters. QUERY is a SQL query in Sybase Transact-SQL syntax. SERVER is the Sybase system to which the QUERY should be forwarded. USER and PASSWORD are respectively the user name and the password which have to be provided to obtain the service of the SERVER. Sybcpl writes the reply from the SERVER to its standard output after doing an on-the-fly conversion to Kleisli's standard exchange format.

The first step in bringing this C program into Kleisli is to wrap it in a simple ML program as follow.

```
fun GetValSybase X = let
  val User = CompObjString.Km(CompObjRecord.ProjectRaw "#user" X)
  val Pwd  = CompObjString.Km(CompObjRecord.ProjectRaw "#password" X)
  val Server = CompObjString.Km(CompObjRecord.ProjectRaw "#server" X)
  val Query  =  CompObjString.Km(CompObjRecord.ProjectRaw "#query" X)
  val (IS, TmpIn) = execute("/mnt/saul/home/khart/pub/entrez/sybcpl",
                                [User, Pwd, Server, Query])
  val _ = close_out TmpIn
  in ("sybcpl " ^ User ^ " " ^ Pwd ^ " " ^ Server ^ " " ^ Query, IS)
  end
```

The ML function `GetValSybase` defined above takes in a Kleisli complex object `X`, which is required to be a record having four fields: `#user`, `#password`, `#server`, `#query`. The values of `X` at these four fields are retrieved into the variables `User`, `Pwd`, `Server`, and `Query`, respectively, using the Kleisli record projection operator `CompObjRecord.ProjectRaw`. These values are converted into native ML strings using the Kleisli string dissembly operator `CompObjString.Km`. Then these four strings are passed on to the C program `sybcpl` via the ML pipe operator `execute`. The result `IS` is returned as a text stream in the standard exchange format of Kleisli.

Sybcpl has been brought into ML in the guise of GetValSybase. However, it is not yet recognized by Kleisli as a new data scanner. The second step is to register it with Kleisli. This is accomplished in ML as follow:

```
val SYBASE = FileManager.ScannerTab.Register(
                GetValSybase,
                Tokenizer.InStreamToTokenStream,
                "SYBASE",
                TypeInput.ReadFromString(
                  "(#user:string,   #password:string," ^
                  " #server:string, #query:string   )" ),
                fn _ => TypeInput.ReadFromString "{''1}")
```

The FileManager.ScannerTab.Register(SCANNER, TOKENIZER, ID, INPUT-TYPE, OUTPUT-TYPE) function is for registering new scanners in Kleisli. SCANNER is expected to be the new data scanner. GetValSybase is the data scanner in this case. TOKENIZER is expected to be a ML function for parsing the text stream returned by SCANNER into Kleisli's token stream. As GetValSybase returns a text stream in Kleisli's standard exchange format, the standard tokenizer Tokenizer.InStreamToTokenStream is used. ID, INPUT-TYPE, and OUTPUT-TYPE are respectively the name, the input type, and the output type of the new scanner to be used in CPL's readfile command. See Section 5.4 for a description of the readfile command.

At this point sybcpl is recognized by Kleisli as the new scanner SYBASE. However, in order to turn it into a full-fledged primitive of CPL, a third step is needed. This step is again done in ML:

```
let val X = Variable.New()
in DataDict.RegisterCooked(
     "Sybase",
     Lambda(X, Apply(ScanObj, Apply(Read(SYBASE, 0), Variable X))),
```

```
        TypeInput.ReadFromString
          "(#user:string, #password:string, #server:string," ^
          " #query:string) -> {''1}")
    end
```

DataDict.RegisterCooked(ID, EXPR, TYPE) is a function for registering a macro definition in Kleisli. ID is the name of the macro. In this case it is Sybase. EXPR is the body of the macro. It has to be an expression in Kleisli's abstract syntax. In this case, the expression is Lambda(X, Apply(ScanObj, Apply(Read(SYBASE,0), Variable X))). Lambda(X, E) is Kleisli's abstract syntax for defining an anonymous function that takes input X and returns E. Read(SCANNER, CHANNEL) is Kleisli's abstract syntax for invoking SCANNER on CHANNEL. In this case, SCANNER is the new scanner SYBASE and CHANNEL is given a dummy value 0. It is unnecessary to worry about this dummy value because Kleisli's query optimizer eventually replaces it with the correct channel number. ScanObj is Kleisli's abstract syntax representing the Kleisli operator for converting a token stream into a complex object. This operator is equivalent to a command to bring an entire database into main memory. It is unnecessary to worry about this apparent inefficiency because Kleisli's optimizer eventually optimizes away this kind of complete loading; see Chapter 6. Thus the whole expression represents a function that takes an input X, uses it as parameters to the SYBASE scanner, scans the specified data into memory, and returns the resulting Kleisli complex object. Since X is used as the input parameter to SYBASE, it is required to be a Kleisli record having four string fields #user, #password, #server, and #query. As SYBASE is expected to return a relational table, the output is expected to be a set containing records of a type to be determined dynamically. TYPE is used to indicate these input-output type constraints.

After the three steps above have been carried out, a new primitive Sybase will be available for use in CPL. Applying this primitive to any record (#user: USER, #password: PASSWORD, #server: SERVER, #query: QUERY) in CPL causes sybcpl USER PASSWD SERVER QUERY to be executed and the result to be returned as a complex object for further manipulation in CPL. It is important to point out that Sybase is now a first-class primi-

tive and can be used freely in any CPL query in any place where an expression of type `(#user:string,#password:string, #server:string,#query:string)-> {''1}`, which is the type specified for `Sybase`, is expected. This is a result of CPL being a fully compositional language, in contrast to SQL which does not enjoy this property.

The new `Sybase` primitive just added to CPL provides us the means for accessing many biological databases stored in Sybase format, including GDB [160] (which is the main GenBank sequence database located at The Johns Hopkins University), SORTEZ [87] (which is a home-brew sequence database located at Penn's genetics department), Chr22DB (which is the local database of the Philadelphia Genome Center for Chromosome 22), etc. These can now be accessed from CPL by directly calling `Sybase` with the appropriate user, password, and server parameters. For convenience and for illustration, I define new primitives for accessing each of them in terms of `Sybase` in CPL as follow. See Chapter 5 for the syntax of CPL.

```
primitive SORTEZ == \Query =>
  Sybase @ (#user:"asn", #password:"bogus",
            #server:"CBILSQL", #query: Query);

primitive GDB == \Query =>
  Sybase @ (#user:"cbil", #password:"bogus",
            #server:"WELCHSQL", #query:Query);

primitive GDB_Tab == \Table =>
  GDB @ ("select * from " ^ Table ^ " where 1=1");

primitive Chr22DB_Tab == \Table =>
   Sybase @ (#user:"guest", #password:"bogus", #server:"CBILSQL",
             #query:"select * from " ^ Table ^ " where 1=1");
```

Hence, for example, SORTEZ takes a string representing an SQL query and passes it via Sybase to the server at Penn's genetics department. Here is a short example for using SORTEZ to look up identifiers from the National Center for Biotechnology Information that are equivalent to the GenBank identifier M15492. This simple query returns a singleton set shown below.

```
primitive CurrentACC == \Id =>
  (SORTEZ @ "select locus, accession, title, length, taxname
             from gb_head_accs
             where pastaccession = '" ^ Id ^ "'");

CurrentACC @ "M15492";

Result: {(#locus: "HUMAREPBG",
          #accession: "M15492",
          #length: 171,
          #taxname: "Homo Sapiens",
          #title: "Human alphoid repetitive DNA repeats 1'' monomer,
                   clone alpha-RI(680) 22-73-I-1.")}
```

Querying ASN.1 databases

The Entrez family of databases is provided by the National Center for Biotechnology Information [150]. This data is stored in ASN.1 format [151], which contains data structures such as sets and records, as well as lists and variants [100] not commonly seen in traditional database models. The use of nested data types make this database non-relational. However, it is easily represented with the native data structures of Kleisli. Unlike the Sybase relational database, there is no existing high-level query language for this database. In order to retrieve Entrez ASN.1 data into Kleisli, it is necessary to design a selection syntax for indexed retrieval of Entrez entries. A mechanism for specifying retrieval of a partial entry

is also added for convenience and efficiency. The resulting program

asncpl -d DATABASE -s SELECTION -p PATH

is written in C by a colleague from Penn's genetics department. It takes three parameters. DATABASE names which Entrez database to use. SELECTION is a boolean combination of index names and values. PATH specifies the part of an entry to be returned. Asncpl retrieves the subset of all DATABASE entries satisfying the SELECTION.

Each database has its own set of index names. Valid indices in the nucleic acid database include `word`, `keyword`, `author`, `journal`, and `organism`. Valid operators for SELECTION are `and`, `or`, and `butnot`. The PATH syntax allows for a terse description of successive record projections, variant selections, and extractions of elements from collections. The formation of the expression can most easily be explained via a traversal of the schema represented as a graph in Figure 9.2. The graph schema is formed first as a tree by placing base types at the leaves, followed by set, lists, records, and variants at the internal nodes, and field and variant labels on the arcs. The tree becomes a graph in this particular schema because there are recursive types present (shown by a dotted line). The PATH expression is built by starting at the root corresponding to the `Seq-entry` type, and building a subtree of the tree while concatenating a dot to the expression for each internal node in the subtree as well as adding arc names as they are encountered.

Based on the above schema, the title and common name of all nucleic acid entries related to human beta globin genes can be extracted by executing the following query:

```
asncpl -d  na
       -s  'gene "beta globin" and organism "homo sapien"'
       -p  Seq-entry{.set.seq-set.}*.seq.descr..(title|org.common)}
```

It causes the following sequence of actions. First, an index lookup is used to retrieve the intersection of entries corresponding to beta globin genes and all the human entries. Then, the path expression is applied to each entry so that only the title and common names

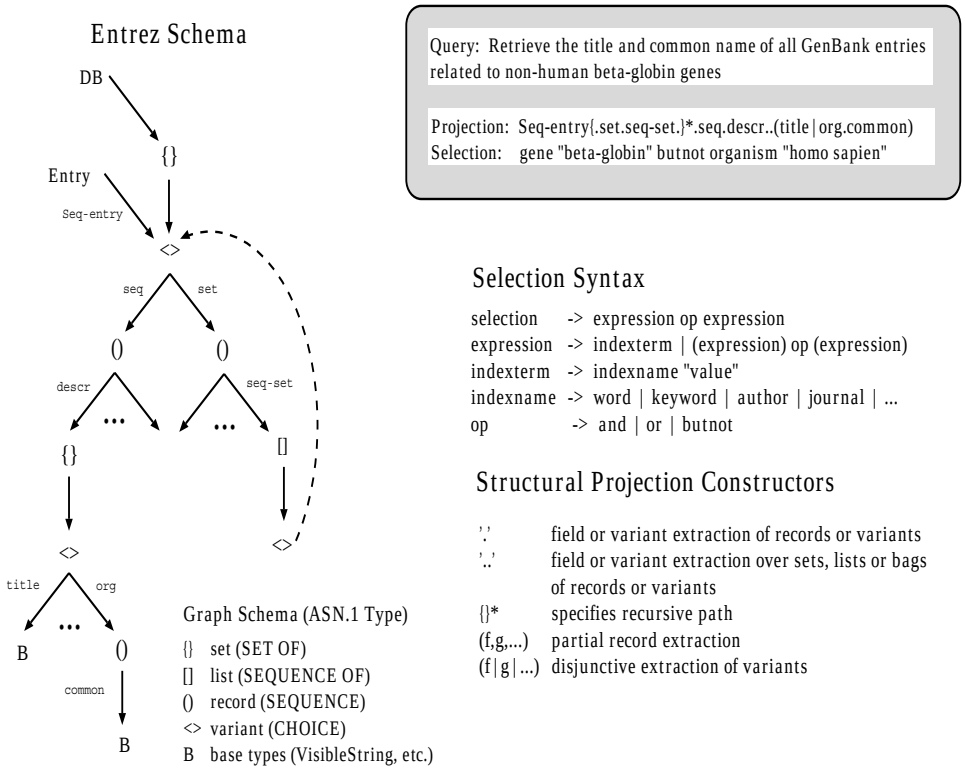


Figure 9.2: Using the ASN.1 server.

are returned. This involves several projection and extraction steps. Only `seq` variant of `Seq-entry` are needed, but a recursive path `{.seq.seq-set.}*` is necessary to specify all of them. A single application of this path selects the `set` variant of `Seq-entry`, projects the field `seq-set`, and then extracts from the resulting list each element that is a `seq`. For each `seq`, the field `descr` is projected and a set of variant types limited by the expression `(title|org.common)` to the strings `title` and the field `common` from the record `org` are returned.

The same three-step procedure is used to bring this C program into Kleisli and CPL. The first step is to wrap it in ML:

```
fun GetValEntrez X = let
  val DB      = CompObjString.Km(CompObjRecord.ProjectRaw "#db" X)
  val Keyword = CompObjString.Km(CompObjRecord.ProjectRaw "#select" X)
  val Path    = CompObjString.Km(CompObjRecord.ProjectRaw "#path" X)
  val (IS, TmpIn) = execute("/mnt/saul/home/khart/pub/entrez/asncpl",
                       ["-d", DB,
                        "-s", StringUtil.stringTrans("'","\"")Keyword,
                        "-p", Path])
  val _ = close_out TmpIn
  in ("asncpl -d " ^ DB ^ " -s " ^ Keyword ^ " -p " ^ Path, IS) end
```

Then asncpl can be accessed from ML via `GetValEntrez`. The second step is to register the latter as a new scanner with Kleisli. As Kleisli supports all of the basic data structures of ASN.1, there is no problem in engineering asncpl so that it outputs in Kleisli's standard exchange format. The registration of the ASN.1 scanner and primitive are done in a similar way as demonstrated with sybcpl. This registration step gives CPL a new primitive `Entrez` that takes in a record (`#db`: DATABASE, `#select`: SELECTION, `#path`: PATH), executes asncpl -d DATABASE -s SELECTION -p PATH, and returns the result as a complex object.

This general primitive brings many databases that have been converted to the National Center for Biotechnology Information's ASN.1 format into CPL including EMBL [95], DDBJ [182], PIR [17], etc. These databases are organized into the three divisions, MEDLINE, nucleotide, and protein. They can now be accessed directly in CPL by calling `Entrez` with the database names `ml`, `na`, and `aa` respectively. Below is a short example of using `Entrez` to find other identifiers corresponding the the accession number `M81409`. Only the first two records in the output are given below. Notice it is a set of sets of variants of records.

```
Entrez @ (#db: "na",
          #select: "accession M81409",
          #path: "Seq-entry.seq.id");

Result:{{<#giim: (#id: 305594, #db: "", #release: "")>,
         <#genbank:(#name:"CEBGLOBIN", #accession:"M81409",
                    #release:"", #version:~1)>},
        {<#genbank:(#name:"M81409_1", #accession:"",
                    #release: "", #version:~1)>,
         <#giim: (#id: 305595, #db: "", #release: "")>},
        ...}
```

Integrating application programs

An important operation performed on biological databases is homologous sequence searching. That is, looking for sequences that are similar. Special application programs are usually used for this purpose. These application programs can also be connected to Kleisli using the same three-step procedure shown earlier. Getlinks is one such program that uses precomputed links in the Entrez family of databases. It is written in C. I only use it in a very simple way in this dissertation:

```
getlinks -n5 -a ACCESSION
```

It looks for the 5 genes most homologous to the one identified by ACCESSION.

The first step is to turn it into a function in ML.

```
fun GetValEntrezAccessionLinks X =
  let val (IS, TmpIn) = execute(
                "/mnt/saul/home/khart/pub/entrez/getlinks",
                ["-n5", "-a", CompObjString.Km X])
      val _ = close_out TmpIn
  in ("getlinks -n5 -a " ^(CompObjString.Km X), IS) end
```

The second step is to register this function as a scanner to Kleisli.

```
val ENTREZLINKS = FileManager.ScannerTab.Register(
                     GetValEntrezAccessionLinks,
                     Tokenizer.InStreamToTokenStream,
                     "ENTREZLINKS",
                     Type.String,
                     fn _ => TypeInput.ReadFromString "{''1}")
```

The third step is to turn the scanner into a full-fledged CPL primitive.

```
let val X = Variable.New()
in DataDict.RegisterCooked(
    "EntrezLinks",
    Lambda(X,Apply(ScanObj,Apply(Read(ENTREZLINKS,0),Variable X))),
    TypeInput.ReadFromString "string -> {''1}")
end
```

It then becomes available for use in CPL. Below is a short CPL query for looking up genes homologous to `CEBGLOBIN`. I display just the first two items in the output.

```
EntrezLinks @ "CEBGLOBIN";

Result: {(#ncbi_id: 173, #linkacc: "M33200", #locus: "HUMHBGAA",
          #title: "Human A-gamma-globin gene, 3' end."),
         (#ncbi_id: 147, #linkacc: "X00424", #locus: "HSGL09",
          #title: "Human gamma-globin gene alternative
                   transcription initiation sites"),
        ...}
```

9.4 Implementing optimization rules in Kleisli

The ability to add new scanners and new primitives to Kleisli does not make it a practical query system. In order to be practical, Kleisli must be able to exploit the capabilities of these new scanners and new primitives. For example, the primitive `Sybase` added in Section 9.3 handles SQL queries. If a CPL query accesses Sybase databases and some operations in that query can be performed directly by the underlying Sybase servers, then Kleisli should try to push these operations to these servers to improve performance. Kleisli has an extensible rule-based optimizer for this purpose. As new primitives are added to Kleisli, new optimization rules should also be added to Kleisli. These rules provide Kleisli with the necessary knowledge to make effective use of these new operators.

The rule base for the optimizer in the core of Kleisli is based on those rules described in Chapter 6. As a consequence of these rules, Kleisli does an aggressive amount of pipelining and seldom generates any large intermediate data. The evaluation mechanism of Kleisli is basically eager. These rules are also used to introduce a limited amount of laziness in strategic places to improve memory consumption and to improve response time.

This core of optimization rules has recently been augmented with a superset of those rules described in Chapter 7. In particular, two join operators have been introduced as additional primitives to the basic Kleisli system. One of them is the blocked nested-loop join [115].

The other is the indexed blocked-nested-loop join where indices are built on-the-fly; this is a variation of the hashed-loop join with dynamic staging [148]. (See Section 9.1 for how the second join operator is implemented in Kleisli.) Both operators have a good balance of memory consumption, response time, and total time behaviors. The former is used for general joins and the latter is used when equality tests in join conditions can be turned into index keys. These two operators are accompanied by over twenty-three new optimization rules to help the optimizer decides when to use them. As my system is fully compositional, the inner relations for these joins can sometimes be subqueries. To avoid recomputation, an operator is introduced to cache the result of selected subqueries on disk. This operator is accompanied by three optimization rules to help the optimizer to decide what to cache.

There are over eight additional optimization rules to make more effective use of the capabilities of `asncpl` by pushing projections and variant analysis on Entrez data from CPL to it. There are over thirteen additional optimization rules to make more effective use of the capabilities of `sybcpl` by pushing projections, selections, and joins on Sybase data from CPL to it. If any relational subquery in CPL only uses relations from the same database and does not use powerful operators, our optimizer is able to push the entire subquery to the server. This capability is a physical realization of Theorem 3.1.4.

This section shows how new optimization rules can be introduced into Kleisli. One of the rules used for pushing joins to `sybcpl` and one of the rules for exploiting `IdxJoin` are presented.

Example: Turning `BlkJoin` into `IdxJoin`

Rewrite rules are expressed in ML by pattern matching on Kleisli abstract syntax objects. Specifically, a rewrite rule R is a ML function that takes in a Kleisli abstract syntax object E and produces a list of equivalent abstract syntax objects [E_1, ..., E_n], where each E_i is a legal substitute for E. Let me reproduce for illustration a rule for turning a blocked nested-loop join into an indexed blocked-nested-loop join given in Section 7.3.

```
fun RuleIdxJoin10(Apply(Primitive BJ, Record R)) =
  if Symbol.Eq(BJ, BlkJoin)
  then case Record.KmTuple(InRec R)
     of [Outer, Pred0, Inner, PredI, Lambda(O, Lambda(I,
         IfThenElse(Eq(E1, E2), E3, False))), Loop]
     => if VarSet.Eq(FreeVar E1, VarSet.Eta O) andalso
           VarSet.Eq(FreeVar E2, VarSet.Eta I)
        then [Apply(Primitive IdxJoin,
                    (Record o OutRec o Record.MkTuple)
                        [Outer, Pred0, Lambda(O,E1),
                         Inner, PredI, Lambda(I,E2),
                         Lambda(O, Lambda(I, E3)), Loop])]
        else []
     | _ => []
  else []
| RuleIdxJoin10 _ = []
```

When this rule is applied to an expression, the following steps take place. In the first step, ML pattern matching is used to check that the expression is a Kleisli abstract syntax object representing the application of a primitive `BJ` to a record `R`. In the second step, the function `Symbol.Eq` provided in Kleisli to check if `BJ` is the blocked nested-loop join operator `BlkJoin`. In the third step, the Kleisli record dissembly operator `KmTuple` is used to inspect the record `R`. This step should return a list `[Outer, Pred0, Inner, PredI, PredI0, Loop]`. `Outer` is the generator of the outer relation of the join, `Pred0` is the filter for the outer relation, `Inner` is the generator of the inner relation, `PredI` is a filter for the inner relation, `PredI0` is the join predicate, and `Loop` is the transformation to be applied to the two records to be joined; see Section 7.2. In this step, ML pattern matching is used to check if `PredI0` is of the form `Lambda(O, Lambda(I, IfThenElse(Eq(E1, E2), E3, False)))`; that is, to check is equality test is part of the join predicate. In the fourth step, the function `VarSet.Eq`

provided in Kleisli is used to check whether `O` is the only free variable in `E1` and whether `I` is the only free variable in `E2`; if this is so, then this equality test can be indexed. In the fifth step, the join predicate is split into `Lambda(O, E1)`, `Lambda(I, E2)`, and `Lambda(O, Lambda(I, E3))`. The first is to be used as the index function. The second is to be used as the probe function. The third is to become the new join predicate. Finally, `BlkJoin` is turned into `IdxJoin`. If any of the steps above fails, the empty list is returned indicating that the rule is not applicable to the given expression.

After a rule is defined in ML, it has to be registered with Kleisli in order for the optimizer to use it. This is done by using the Kleisli function `RuleBase.Reductive.Add(`THRESHOLD, NAME, RULE`)` or the Kleisli function `RuleBase.Nonreductive.Add(`THRESHOLD, NAME, RULE`)`. In both cases, NAME is a string used for identifying the rule within Kleisli, RULE is the ML function implementing the rule, and THRESHOLD is the firing threshold of the rule. The difference between these two add functions is that the former adds the rule as a reductive rule while the latter adds it as a nonreductive rule.

More detail of Kleisli is needed to explain these two types of rules. Kleisli divides its rule base into two parts: reductive rules and nonreductive rules. It assumes that the reductive rules form a strongly normalizing rewrite system and that normal forms are always better than non-normal forms. Most of the rules given in Chapters 6 and 7 are reductive rules. It makes no assumption on nonreductive rules. An example of nonreductive rule is the commutative rule *if* e_1 *then* (*if* e_2 *then* e_3 *else* e_4) *else* $e_4 \rightsquigarrow$ *if* e_2 *then* (*if* e_1 *then* e_3 *else* e_4) *else* e_4 mentioned in the beginning of Chapter 7.

Given an expression to be optimized, Kleisli applies the reductive rules repeatedly until a normal form is reached; rules with lower threshold are given precedence over rules with higher threshold. It then applies the nonreductive rules in all possible ways to generate more alternatives. An alternative is discarded if its cost computed based on the currently active cost function exceeds the current best alternative by more than a specified hill-climbing factor. Kleisli then repeats the optimization cycle with each remaining alternative in a best-first manner [202]. The process stops when no new alternative is generated or when

a specified time limit is reached. The best alternative is then picked. In comparison to a sophisticated optimizer generator like that of Exodus [74], Volcano [76], or Starburst [85], this optimizer is simple minded and there is room for improvement. In addition, Kleisli can be flagged to present all good alternatives to the user so that he can make the final choice; this feature can be important when the cost function provided is not sufficiently refined.

Kleisli allows a prologue phase to be applied to an alternative before the reductive rules are applied and an epilogue phase to be applied to a normal form before it is stored as an alternative optimized query. These two phases can be used to invoke alternative specialized optimizers that a sophisticated programmer may want to used in conjunction with Kleisli's optimizer. They can also be used to make certain rules easier to implement. For example, some rules in Chapter 7 requires unique natural numbers to be generated; these numbers are best generated during the epilogue phase.

The ML function `RuleIdxJoin10` is registered in my system as a reductive rule. Its effect can be seen in the optimizer output for experiment H in Section 8.3.

```
RuleBase.Reductive.Add(1100, "IdxJoin:IdxJoin10:", RuleIdxJoin10);
```

Example: Pushing joins to `sybcpl`

Let me reproduce one of the rules used for migrating blocked nested-loop joins from Kleisli and CPL to Sybase servers. It is an ML function that takes a Kleisli abstract syntax object and produces a list of equivalent objects.

```
fun RulePush3(Apply(Primitive BJ, Record R)) =
 if Symbol.Eq(BJ, BlkJoin)
 then case Record.KmTuple(InRec R)
     of [Lambda(A, Apply(Read(Scanner0, _), DB0)), Pred0,
         Lambda(B, Apply(Read(ScannerI, _), DBI)), PredI,
         PredI0, Loop]
```

```
      => if Scanner.Eq(ScannerO, SYBASE) andalso
            Scanner.Eq(ScannerI, SYBASE) andalso
            OkayToPush2(DBO, DBI, PredIO)
         then let val (Join, Pred) = FindJoinCond2(DBO, DBI, PredIO)
                  val DB     = CombineSQL(Join, DBO, DBI)
                  val D      = Variable.New()
                  val RO     = Reformat(DBO, DBI)
                  val RI     = Reformat(DBI, DBO)
                  val O      = Apply(RO, Apply(ScanObj, Variable D))
                  val I      = Apply(RI, Apply(ScanObj, Variable D))
                  val Pred   = Apply(Apply(Pred, O), I)
                  val Loop   = Apply(Apply(Loop, O), I)
                  val O      = Apply(PutObj,Apply(RO,
                                   Apply(ScanObj,Variable D)))
                  val I      = Apply(PutObj,Apply(RI,
                                   Apply(ScanObj,Variable D)))
                  val PredO  = Apply(PredO, O)
                  val PredI  = Apply(PredI, I)
              in [Apply(Primitive PreJoin,
                        Record(OutRec(Record.MkTuple[
                          Lambda(A, Apply(Read(SYBASE, O), DB)),
                          Lambda(D, IfThenElse(PredO,
                            IfThenElse(PredI,Pred,False),False)),
                          Lambda(D, Loop)])))]
              end
         else []
       | _ => []
    else []
  | RulePush3 _ = []
```

When this rule is applied to an expression, the following things happen. In the first step, ML pattern matching is used to check if the expression is the application of a primitive `BJ` to a record `R`. In the second step, the `Symbol.Eq` function provided in Kleisli is used to check if the primitive `BJ` is the blocked nested-loop operator `BlkJoin` described in Section 7.2. In the third step, the record dissembly operator `Record.KmTuple` provided in Kleisli is used to inspect the record `R`. This step should return a list `[Outer, Pred0, Inner, PredI, PredI0, Loop]`. `Outer` is the generator of the outer relation of the join. `Pred0` is a filter for the outer relation. `Inner` is the generator of the inner relation. `PredI` is a filter for the inner relation. `PredI0` is the join condition. `Loop` is the transformation to be applied to the two records to be joined. In the fourth step, the function `Scanner.Eq` provided in Kleisli is used to check if `Outer` and `Inner` are both producing data using the `SYBASE` scanner. In the fifth step, it checks whether the two relations being scanned are on the same server and whether the join condition can be pushed to `Sybase`. This task is accomplished by a simple function `OkayToPush2`, which I have to define in ML. In the sixth step, the join condition `PredI0` is split into a pair `(Join, Pred)` using the function `FindJoinCond2`, which I also have to define in ML. `Join` is the part of `PredI0` that can be pushed to `Sybase`, while `Pred` is the remainder of `PredI0` which cannot be pushed due to presence of powerful operators. In the seventh step, a new SQL query `DB` is formed using the function `CombineSQL` provided in a Kleisli library. `DB` is formed by pushing `Join` to join the outer and inner relations. (This transformation is a conceptually simple rewrite step that can be illustrated as follows. Suppose the outer relation is the query select A from B where C and the inner relation is the query select D from E where F. Then `CombineSQL` produces select A, D from B, E where C and F and Join, with some renamings if necessary.) In the eighth step, `Pred0`, `PredI`, `Pred`, and `Loop` have to be adjusted because the data coming in has changed. (As can be seen from the example, the data now come from a single relation with columns A, D, as opposed to from two relations with column A and column D.) A function `Reformat` is written in ML to accomplish the task of extracting the right fields from the new input. The adjusted versions of `Pred0`, `PredI`, `Pred`, and `Loop` are obtained by applying the originals to the reformatted data. Finally, these modified fragments are recombined into a `Prejoin`, which is the simple filter loop described in Section 7.2.

After this ML function is defined, it has to be registered with the Kleisli optimizer rule base as shown in the piece of ML program below. Then it is automatically used by the optimizer to convert blocked nested-loop joins in CPL to joins in `Sybase`.

```
RuleBase.Reductive.Add(120, "PushGDB3:Push3:", RulePush3);
```

Below is a short example illustrating the kind of optimizations that this system does. This CPL query joins three Sybase relations.

```
primitive Loci22 == { (#locus_symbol: x, #genbank_ref: y) |
  (#locus_symbol:\x,
   #locus_id:\a, ...) <- GDB_Tab @ "locus",
  (#genbank_ref:\y,
   #object_id:a,
   #object_class_key:1,...) <- GDB_Tab @ "object_genbank_eref",
  (#loc_cyto_chrom_num:"22",
   #locus_cyto_location_id:a,...) <-GDB_Tab @"locus_cyto_location"};
```

The optimizer is able to migrate all the selections, projections, and joins in the above query completely to the Sybase server, resulting in the optimized version shown below. See also the sample optimizer output for experiment K given in Section 8.5.

```
primitive Loci22 ==
    GDB @ "select locus_symbol, genbank_ref
            from locus, object_genbank_ref, locus_cyto_location
            where locus.locus_id = locus_cyto_location_id
            and locus.locus_id = object_genbank_eref.object_id
            and object_class_key = 1 and loc_cyto_chrom_num = '22'";
```

9.5 Two biological queries in CPL and a manifesto

Having connected Kleisli to several biological data sources, it is then possible to manipulate information from these data sources using the high-level query language CPL. This section contains two simplified examples taken from real biological queries that were posed to my system when it first became operational. I close this chapter with a short manifesto on querying heterogenous biomedical data sources.

There are several things about these queries that are worth pointing out. First, these examples require several different data sources to be accessed. Second, data from these different sources are freely combined in CPL without any special handling. Third, the last example requires non-flat output — it needs three levels of nesting in order to group the output correctly. Fourth, their implementation in CPL are all short and concise. Fifth, all the examples use databases gigabytes in size but all of them are completed within minutes. This performance is within striking distance of hand-coded programs but without the sweat. Finally, the last example is a template for dealing with several of the hard queries listed in a Department of Energy report [57]. This is indicative of the potential of Kleisli and CPL.

Example: Find chromosome 22 sequence tag sites in GDB but not in Chr22DB

Sequence tag sites currently in use in Chr22DB are found using the following CPL query. (See Davidson, Kosky, and Eckman [54] for a primer written for database workers on terminology used by biologists.)

```
primitive STSinUSE == { n |
   (#name:\n,
    #lab_code:"GDB",
    #item:"STS",
    #printname:"Y", ...) <- Chr22DB_Tab @ "names"};
```

After this primitive is defined, the desired query can be implemented by taking the difference between it and `Loci22` in CPL.

```
{x.#locus_symbol | \x <- Loci22} setdiff STSinUSE;
```

Example: Find annotation information on known DNA sequences on human chromosome 22 as well as information on sequences homologous to them

This query is the same example given in Section 1.5. It needs a CPL subquery `Homologs` which takes in a GDB identifier, looks up the equivalent identifiers and other information in SORTEZ, and then applies `EntrezLinks` to find similar sequences.

```
primitive Homologs == \Id =>
   {(y, EntrezLinks @ (y.#accession)) | \y <- CurrentACC @ Id};
```

Then a simple comprehension over `Loci22` accomplishes the task in CPL.

```
{(x, Homologs @ (x.#genbank_ref)) | \x  <- Loci22};
```

Manifesto

In Spring 1993, the Department of Energy [57] published a report, listing twelve queries that were claimed to be unanswerable "until a fully relationalized sequence database is available." Some of these queries require further interpretation of source data, but the majority can be answered on the basis of existing source data. Presumably these were thought to be impossible because they involve the integration of databases, structured files, and applications — something well beyond the capabilities of any existing heterogenous database system.

Kyle Hart from Penn's genetics department has been able to implement these queries using the open query system Kleisli and the collection programming language CPL that serves as Kleisli's high-level query language. The last example is a template solution for many of these queries. The strength of my system derives from my novel approach to languages for structured data that greatly expands the expressive power of database query languages. As sketched in the preceding sections, the current system provides transparent access to biological data sources including relational databases, non-standard (structured file) databases, and application programs. It can freely combine information from these heterogenous sources; it incorporates a rule base that can exploit optimization techniques in these sources; and its pool of external data scanners and data writers can be readily expanded to connect to new data sources.

One very important feature of CPL is that it is fully compositional. This feature has obvious benefits as a programming language, but more broadly, it gives us the capability of defining user views simply in terms of queries. These views in turn can be used in other views. Once documented to reflect their interpretation, such views can be used to provide relevant, succinct, and comprehensible information to users at various levels of sophistication.

This new approach to database languages may call into doubt the necessity or advisability of building monolithic databases for biological data. Individual groups, rather, can simply publish their data schema along with a query interface to the data. Tools such as CPL and Kleisli (together with schema restructuring tools such as that developed by Davidson, Kosky, and Eckman [54]) can then be used to reconcile the schema differences, create distributed views, and retrieve integrated information.

Part IV

The perspective of a logician-engineer

Chapter 10

Conclusion and Further Work

The final test of a theory is its capacity to solve the problems which originated it. George Dantzig

The first part of this dissertation begins in Chapter 1 with the belief that structural recursion is a useful database programming paradigm and ends in Chapter 5 with a concrete query language for nested collections with many desirable properties. In the course of these five chapters, I have examined the expressive power of $\mathcal{NRC}(\mathbb{B}, =)$ and its many practical extensions. At one end of the spectrum is $\mathcal{NRC}(\mathbb{B}, =)$, which is classical because its queries are generic and internal [98]. At the other end of the spectrum is $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq)$ which is much closer in strength to a real query language such as SQL, because it has arithmetic, orderings, and aggregate functions. The second part of this dissertation begins in Chapter 5 with the design of a real query language and ends in Chapter 9 with the use of an extensible query system for querying heterogenous data sources. In the course of these five chapters, I have touched on the topics of language design, query optimization, openness, and have implemented a working prototype of Kleisli and CPL. This chapter offers a summary of the major contributions of this work, an explanation of its relationship to other approaches to querying databases, and a list of future projects.

ORGANIZATION

Section 10.1. The major contributions of this dissertation in the theory, practice, and application of querying nested collections are summarized. It is hoped that this summary conveys some of the merits of my approach to querying nested collections.

Section 10.2. There are several alternative approaches to generalizing flat relational databases. I briefly examined them in this section. In particular, I explain where my approach lies in relationship to them.

Section 10.3. I believe that the most fruitful directions for future work lies in the investigation of new collection types that are useful in real applications. This section identifies what I believe are the more fascinating possibilities.

10.1 Specific contributions

This dissertation proposes a new paradigm for the design, study, and implementation of query languages. The paradigm is to organized query languages around a restricted form of structural recursion. I believe that this approach to querying nested collections is rich, interesting, general, and practical. Many contributions have been made in the theory, practice, and application of query languages for nested collections. I hope the list below, of some of these contributions, conveys some of the merits of my approach.

- The relationship of this restricted form of structural recursion to relational languages is established in Chapter 2. $\mathcal{NRC}(\mathbb{B},=)$ obtained by imposing my restricted structural recursion on sets is equivalent to several classical nested relational languages.
- The scalability of the basic language $\mathcal{NRC}(\mathbb{B},=)$ is shown by extending it with arithmetic, aggregate functions, and orders in Chapter 3; with lists, bags, and variants in Chapter 5; with token streams in Chapter 6; and with external functions in Chapter 9.

- The conservative extension property, useful in understanding the expressive power of query languages, is studied in Chapter 3. A general technique based on the equational axioms arising from my restricted form of recursion is introduced for proving the conservative extension property.

- $\mathcal{NRC}(\mathbb{B}, =)$ and its many extensions are shown in Chapter 3 to possess the conservative extension property. The conservative extension result in the presence of the *powerset* operator is quite surprising.

- The finite-cofiniteness property, useful in understanding the limitations of query languages, is studied in Chapter 4. A general technique based on the conservative extension property is introduced for proving the finite-cofiniteness property.

- $\mathcal{NRC}(\mathbb{B}, \mathbb{Q}, +, \cdot, -, \div, \sum, =, \leq^{\mathbb{Q}})$ is shown in Chapter 4 to be finite-cofinite on certain classes of graph queries. This result uniformly extends many well-known results on flat relational calculus to a language that is closer in strength to SQL. It also settles several conjectures on a popular bag query language.

- A high-level query language, CPL, based on expressing my restricted form of recursion using the comprehension syntax is designed in Chapter 5. Also variable-as-constant patterns are used for the first time in pattern matching in a query language.

- A prototype extensible query system, Kleisli, organized around my restricted form of structural recursion is built in Chapter 9. CPL is implemented on top of it and serves as it high-level query language.

- Techniques for doing an aggressive amount of pipelining in languages organized around my restricted form of recursion, to reduce memory consumption and to improve response time, are shown in Chapter 6. These pipelining techniques have been implemented in my prototype and tested in Chapter 8.

- Ways for generalizing many classical optimizations to languages organized around my restricted form of recursion are shown in Chapter 7. These techniques have been implemented in my prototype and tested in Chapter 8.

- Kleisli and CPL are used for querying nested collections in a general way. They proved satisfactory in querying many biological data sources in Chapter 9.
- The implementation of the prototype contains approximately twenty three thousand lines of ML codes and took approximately two man-months to develop. This prototype is a substantial contribution to showcase the use of functional programming languages in rapid prototyping and in serious applications.

10.2 A Gestalt

Flat relational systems have to be stretched and modified in two directions to satisfy the needs of modern database applications. The first direction is to have a richer data model than flat tables and the second direction is to have a more expressive query language than flat relational algebra.

Past and present effort in creating better databases can broadly be classified into three alternatives. The first alternative is focused on making the data model richer; the development of nested relational databases falls into this category. The second alternative is focused on making the query language more powerful; the development of deductive databases falls into this category. The third alternative uses more powerful data model as well as more powerful query languages; the development of object-oriented databases falls into this category.

Let me describe these three lines of development and try to relate my work to them.

Nested relational databases

allow the components of tuples in a relation to be relations. The data model is therefore more natural for certain problems, such as the salary history example of Makinouchi [141]. The better known proposals for nested relational databases are those of Thomas and Fischer [183], Schek and Scholl [169], and Colby [45]. The following comments can be made.

- They did not take into account of modern and useful data types such as variants [100], bags, and until recently [46], lists.
- Their development was strongly tied to sets. For this reason, it is not easy to extend them in a uniform manner to include the new data types mentioned above.
- Their development followed a trend of increasing semantic complexity without a corresponding increase in modeling power and expressive power.
- As discussed in Chapter 2 and in the proof of Proposition 2.5.3, important query language concepts such as orthogonality and mapping of functions are missing from them.

Deductive databases

introduce a fixpoint operator into the first-order logic of flat relations. Expressive power is greatly increased by their ability to compute recursive queries. The early theory of deductive databases was most clearly described by Lloyd and Topor [136]. The most notable work on their development is the large body of knowledge gathered on the optimization of recursive rules [191, 164, 193, 94, 86]. The following additional comments can be made on deductive databases.

- The basic data type used in various versions of Datalog, the main query language for deductive databases, is still the flat relations. Therefore the disadvantages observed by Makinouchi [141] on the flat relational data model applies.
- The termination of fixpoint evaluation is guaranteed for pure Datalog. This property is destroyed in the presence of operations such as addition and multiplication, which are necessary in real applications.
- Judging from the variety of semantics [71, 190, 120, 192] for negation in Datalog, there is still no agreement on a general treatment of negation in the presence of fixpoint.

- There have been attempts to enhance datalog to deal with sets, most notably Kuper [122] and Naqvi and Tsur [149]. It remains to be seen how bags and lists fit into the picture.

Object-oriented databases

essentially turn object-oriented programming languages into database systems. So they have powerful data models and are very expressive. There are many working prototypes and systems. Some of the better known examples include ORION [116, 114], O_2 [58], Exodus [37], IRIS [201], GemStone [35], and ObjectStore [126]. The following comments can be made.

- The diversity of object-oriented database systems is bewildering. This diversity is not surprising, as they took as their starting points object-oriented languages that are very different.
- There is much system-building effort but little theoretical output. In particular, the behavior of these systems tends to be defined by implementation. This can perhaps be attributed to the fact that many foundational issues in object-oriented languages are still in flux. See Gunter and Mitchell [81]; Cook [48]; Borning [25]; Cook, Hill, and Canning [49]; etc.
- These systems can do everything, provided the user works in his host language such as C++ [177] or Smalltalk [73]. With a few exceptions such as O_2 [16], they generally lack true query languages.
- They generally support some sort of sets, bags, and lists. But they generally do not support all of them in a uniform way.

In constrast to the above languages and systems, CPL cleanly and uniformly supports lists, bags, sets, and potentially more collection types. CPL has more flavor of the nested relational and the object-oriented approaches than the deductive approach because CPL shares with the former a richness in their data models not found in the latter. CPL is more radical than the nested relational approach and is less radical than the object-oriented approach. The following additional comments can be made on its connections to these alternatives.

- CPL restricted to $\mathcal{NRC}(\mathbb{B}, =)$ is equivalent in strength to a well-known nested relational algebra; see Theorem 2.5.3. However, this dissertation is ample evidence that CPL comes with a more flexible and more general theory.

- CPL restricted to $\mathcal{NRC}(\mathbb{B}, =)$ but augmented with a bounded fixpoint operator is equal in strength to Datalog with negation over queries on flat relations; see Suciu [178]. However, CPL is more robust and more practical in the following sense. If numbers and the basic arithmetic operations are added to the former, it remains very much the same language. On the other, the semantics of Datalog with negation can get drastically changed by these additions; for instance, termination is no longer guaranteed.

- Neither I nor my colleagues have attempted a formal comparison of CPL to any object-oriented system. I do not think such a comparison is possible given the current state of affairs of object-oriented database systems. Nevertheless, let me make two remarks. As far as data model is concerned, if one strips away the more poorly understood features of object-oriented data models, then CPL can be made as rich as any of them by adding either a suitable notion of identifiers or recursive values. As far as expressive power is concerned, CPL cannot match them since they use full-fledged programming languages. However, recall from Chapter 1 that CPL is obtained by imposing a strong restriction on structural recursion. So one can recover for CPL some extra horsepower by relaxing the restriction.

Perhaps what is most remarkable in this compressed account is the fact that there is a formal connection between CPL and nested relational algebra and Datalog at all, given that their starting points are so different.

10.3 Further work

It is customary to end a dissertation with a list of future projects. There are many projects that I can propose as future work, especially in improvements to the prototype. However, I think such projects are best left to the engineer. Instead, I want to put forward possibilities which are more speculative.

In this dissertation I have focused on reporting my results on query languages for sets. I have also worked on orsets and bags. My work with Libkin [131] on orsets does not have an impact on this dissertation. However, it has lead to important advances on the study of disjunctive and partial information [129]. My work with Libkin [135] on bags does have an impact on this dissertation. In fact, most of the results on aggregate functions in Chapter 3 and Chapter 4 were originally developed to answer questions on bag queries.

From this experience, I believe that one of the most fruitful direction for future work will be the study of new collection types. Real world applications are without doubt the best source of inspiration for new collection types. So let me close this dissertation by listing some of the more fascinating ones.

Indexed collections as first-class citizens

The use of indices is a very important factor in the performance of flat relational databases. Traditionally, indices on a relation are recorded separately from the relation. This scheme was intended to separate implementation from semantics. But is such a scheme scalable to nested relational databases? What if I want to have a set of sets where the outer set is not indexed but each member of it are independently indexed? What if I want to have

an even more complex organization of indices? I think it is possible to introduce explicitly indexed collections into a query language as a first-class citizen with its own type and expression constructs, without messing up the separation of implementation and semantics of the query language. In fact, I believe such an approach will exhibit an orthogonality which can simplify the theoretical study and the practical treatment of indexed collections.

Arrays as a special case of indexed collections

Arrays are one of the most exciting collection types. They are certainly the earliest collection type to be incorporated into programming languages for they are present in FORTRAN, albeit in a very primitive way. They become more sophisticated in APL [107] and even more so in Sisal [65]. However, as lamented by Maier and Vance [140], they have been ignored in query languages. Buneman [32] recently discussed the fast Fourier transform as a database query. He chose most of his operators for reason of expedience. However, he did choose a particularly striking construct for accessing arrays: $\{e \mid x_i \in A\}$ for binding x and i respectively to an element and its position in the array A. This same choice was later copied by Fegaras in a more general paper [64]. I think this idea of binding both element and position will be a fundamental feature of query languages with arrays as first-class citizen. I think it can be generalized to binding element and index value in the case of indexed collections as first-class citizen.

Recursive types

Consider the problem of modeling cities and states: Each city record should have a field indicating the state in which the city is located and each state record should have a field indicating all the cities located in that state. Without knowledge about keys, it is very hard to model this in a relational database. It is easy to model this in an object-oriented database using object identifiers. However, using system-specific identifiers leads to transportability problem. It has been noted [14] for quite some time that it is not easy to move objects from one object-oriented database to another because object identifiers that make sense in

the first database are not going to make sense in second. Regular trees [50] can be used to model problems such as cities and states. Since they do not use the notion of identifiers, it is easier to transport them across databases. So they may be a good alternative. However, regular trees are generally manipulated using full-fledged recursion. I believe it is possible to impose some restrictions to make regular-tree programming recursionless, or at least more controlled.

Display abstraction and hypertext

I have assumed that when a set is printed, all its members are printed in their entirety. However, in a more advanced interface, it might be better to hide some detail. For example, if the output is connected to a hypertext device on the World Wide Web [20], it may be better to hide some detail and to incrementally expose it when various hyperlinks are followed. One can of course first compute the output completely and then do the hiding while preparing the hypertext pages. Alternatively, one can delay computing the detail until its hyperlink is followed. In this way, work is not wasted if the hyperlink is not followed. It will be challenging to see how a hypertext device can be abstracted away and how delays can be propagated.

Bibliography

[1] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In S. Abiteboul, P. C. Fisher, and H.-J. Schek, editors, *LNCS 361: Nested Relations and Complex Objects in Databases*, pages 117–138. Springer-Verlag, 1989.

[2] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[3] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proceedings of 19th International Conference on Very Large Databases*, 1993.

[4] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[5] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, England, 1987.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[7] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings 6th Symposium on Principles of Programming Languages, Texas, January 1979*, pages 110–120, 1979.

[8] J. Albert. Algebraic properties of bag data types. In *Proceedings of 17th International Conference on Very Large Databases*, pages 211–219, 1991.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[10] Malcolm Atkinson, Philippe Richard, and Phil Trinder. Bulk types for large scale programming. In J. W. Schmidt and A. A. Stogny, editors, *LNCS 504: Next Generation Information System Technology*, pages 229–250, Berlin, 1990. Springer-Verlag.

[11] AT&T Bell Laboratories, Murray Hill, NJ 07974. *Standard ML of New Jersey User's Guide*, February 1993.

[12] L. Augustsson. A compiler for lazy ML. In *Symposium on LISP and Functional Programming*, pages 218–227, Austin, Texas, 1984.

[13] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[14] F. Bancilhon. Object-oriented database systems. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 152–162, Los Angeles, California, 1988.

[15] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. A powerful and simple database language. In *Proceedings of International Conference on Very Large Data Bases*, pages 97–105, 1988.

[16] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Proceedings of 2nd International Workshop on Database Programming Languages*, pages 122–138. Morgan Kaufmann, 1989.

[17] W.C. Barker, D.G. George, L.T. Hunt, and J.S. Garavelli. The PIR protein sequence database. *Nucleic Acids Research*, 19:2231–2236, 1991.

[18] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Series in Computer Science. Prentice Hall International, New York, 1990.

[19] Catriel Beeri and Yoram Kornatzky. Algebraic optimisation of object oriented query languages. *Theoretical Computer Science*, 116(1):59–94, August 1993.

[20] Tim Berners-Lee. The World Wide Web initiative: The project. Available via `http://info.cern.ch/hypertext/WWW/TheProject.html` on WWW.

[21] R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6:159–189, 1986.

[22] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, Volume F36 of *NATO ASI Series*, pages 3–42. Springer-Verlag, 1987.

[23] R. S. Bird and P. Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice-Hall International, 1988.

[24] D. Bjorner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988. Proceedings of IFIP TC2 Workshop, Gammel Avernaes, Denmark, October 1987.

[25] A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 30–46, 1986.

[26] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[27] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *LNCS 193: Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40. Springer-Verlag, 1985.

[28] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[29] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[30] O. P. Buneman, R. Nikhil, and R. E. Frankel. An implementation technique for database query languages. *ACM Transactions on Database Systems*, 7(2):164–187, June 1982.

[31] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[32] Peter Buneman. The fast Fourier transform as a database query. Technical Report MS-CIS-93-37/L&C 60, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, March 1993.

[33] Peter Buneman, Kyle Hart, and Limsoon Wong. Answering some "unanswerable" biological queries, March 1994. Presented at ACM Workshop on Information Retrieval and Genomics, Bethesda, May 1994. Available via `http://www.cis.upenn.edu/~wfan/DBHOME.html` on WWW.

[34] R. M. Burstall, D. B. Macqueen, and D. T. Sanella. HOPE: An experimental applicative language. In *Proceedings of 1st LISP Conference*, pages 136–143, Stanford, California, 1980.

[35] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[36] Luca Cardelli. Types for data-oriented languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *LNCS 303: Advances in Database Technology — International Conference on Extending Database Technology, Venice, Italy, March 1988*. Springer-Verlag, 1988.

[37] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for Exodus. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, 1988.

[38] Ashok Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[39] Ashok Chandra and David Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 1:1–15, 1985.

[40] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 59–70, Washington, D. C., May 1993.

[41] E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, June 1970.

[42] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[43] E. F. Codd. Fatal flaws in SQL, part I. *Datamation*, pages 45–48, 15 August 1988.

[44] E. F. Codd. Fundamentals ignored in SQL: The resulting inadequacies, part 1. *The Relational Journal*, 3(1), 1991.

[45] Latha S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

[46] Latha S. Colby. *Query Languages and a Unifying Framework for Non-traditional Data Models*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana 47405-4101, May 1993. Available as Indiana University Computer Science Technical Report 381.

[47] Mariano P. Consens and Alberto O. Mendelzon. Low-complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116:95–116, 1993.

[48] W. Cook. Object-oriented programming versus abstract data types. In *LNCS 489: Foundations of Object-Oriented Languages*, pages 151–178. Springer-Verlag, 1991.

[49] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, January 1990.

[50] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[51] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.

[52] C. J. Date. A critique of the SQL database language. *SIGMOD Record*, 14(3):8–52, November 1984.

[53] C. J. Date. Some principles of good language design. *SIGMOD Record*, 14(3):1–7, November 1984.

[54] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. Technical Report MS-CIS-93-94/L&C 74, University of Pennsylvania, Philadelphia, PA 19104, December 1993.

[55] Anuj Dawar. *Feasible Computation Through Model Theory.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, May 1993. Available as UPenn Technical Report MS-CIS-93-51.

[56] Jan Van den Bussche. Complex object manipulation through identifiers: An algebraic perspective. Technical Report 92-41, University of Antwerp, Department of Mathematics and Computer Science, Universiteitsplein 1, B-2610 Antwerp, Belgium, September 1992.

[57] Department of Energy. *DOE Informatics Summit Meeting Report*, April 1993. Available via gopher at `gopher.gdb.org`.

[58] O. Deux. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[59] Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, San Diego, 1972.

[60] Martin Erwig and Udo W. Lipeck. A functional DBPL revealing high level optimizations. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, pages 306–321. Morgan Kaufmann, August 1991.

[61] R. Fagin. Finite model theory — a personal perspective. *Theoretical Computer Science*, 116(1):3–32, August 1993.

[62] Joseph H. Fasel, Paul Hudak, Simon Peyton-Jones, and Philip Wadler. The functional programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[63] Leonidas Fegaras. Efficient optimization of iterative queries. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 200–225. Springer-Verlag, January 1994.

[64] Leonidas Fegaras. A uniform calculus for bulk data types, February 1994. Manuscript available from `fegaras@cse.ogi.edu`.

[65] John T. Feo. Arrays in Sisal. In Lenore Mullin, Michael Jenkins, Gaetan Hains, Robert Bernecky, and Guang Goa, editors, *Arrays, Functional Languages, and Parallel Systems*, pages 93–106. Kluwer, Boston, 1991.

[66] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, England, 1988.

[67] M. A. Firth. *A Fold/Unfold Transformation System for a Nonstrict Language*. PhD thesis, Department of Computer Science, University of York, Heslington, York, Y01 5DD, England, December 1990.

[68] Johann Christoph Freytag. *LNCS 261: Translating Relational Queries into Iterative Programs*. Springer-Verlag, Berlin, 1987.

[69] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial time hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984.

[70] Haim Gaifman. On local and non-local properties. In *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, pages 105–135. North Holland, 1982.

[71] M. Ginsberg. *Nonmonotonic Reasoning.* Morgan Kaufmann, Los Alto, California, 1988.

[72] A. Goldberg and R. Paige. Stream processing. In *Proceedings of ACM Symposium on LISP and Functional Programming*, pages 53–62, Austin, Texas, August 1984.

[73] A. Goldberg and D. Robson. *Smalltalk80: The Language And Its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[74] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, November 1987. Available as University of Wisconsin Computer Sciences Technical Report 724.

[75] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[76] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, and Richard H. Wolniewicz. Extensible query optimization and parallel execution in Volcano. In Johann Christoph Freytag, David Maier, and Gottfried Vossen, editors, *Query Processing for Advanced Database Systems*, Chapter 11, pages 305–335. Morgan Kaufmann, San Mateo, California, 1994.

[77] Stephane Grumbach and Tova Milo. Towards tractable algebras for bags. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 49–58, Washington, D. C., May 1993.

[78] Stephane Grumbach, Tova Milo, and Yoram Kornatzky. Calculi for bags and their complexity. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 65–79. Springer-Verlag, January 1994.

[79] Stephane Grumbach and Victor Vianu. Playing games with objects. In S. Abiteboul and P. C. Kanellakis, editors, *LNCS 470: 3rd International Conference on Database Theory, Paris, France, December 1990*, pages 25–39. Springer-Verlag, 1990.

[80] Dirk Van Gucht and Patrick C. Fischer. Multilevel nested relational structures. *Journal of Computer and System Sciences*, 36:77–105, 1988.

[81] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, 1994.

[82] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[83] Marc Gyssens and Dirk Van Gucht. The powerset algebra as a result of adding programming constructs to the nested relational algebra. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 225–233, Chicago, Illinois, June 1988.

[84] Marc Gyssens and Dirk Van Gucht. A comparison between algebraic query languages for flat and nested databases. *Theoretical Computer Science*, 87:263–286, 1991.

[85] L. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[86] R. W. Haddad and J. F. Naughton. Counting methods for cyclic relations. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 333–340, 1988.

[87] K. Hart, D. B. Searls, and G. C. Overton. SORTEZ: A relational translator for NCBI's ASN.1 database. *Computer Applications in the Biosciences.* To appear. See also UPenn Technical Report CBIL-9203.

[88] Kyle Hart. *SORTEZ Reference Manual.* University of Pennsylvania School of Medicine, Department of Genetics, Computational Biology and Informatics Laboratory, Philadelphia, PA 19104, 1992. Available as UPenn Technical Report CBIL-9203.

[89] Kyle Hart and Limsoon Wong. A brief introduction to Kleisli, an open query system in ML, January 1994. Manuscript available from `limsoon@saul.cis.upenn.edu`.

[90] Kyle Hart and Limsoon Wong. CPL as a query language for genetic databases, January 1994. Manuscript available via `http://www.cis.upenn.edu/~wfan/DBHOME.html` on WWW.

[91] Kyle Hart and Limsoon Wong. A query interface for heterogenous biological data sources, February 1994. Manuscript available via `http://www.cis.upenn.edu/~wfan/DBHOME.html` on WWW.

[92] P. Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional Programming and Its Applications*, pages 177–192. Cambridge University Press, 1982.

[93] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, Chichester, England, 1990.

[94] L. J. Henschen and S. A. Naqvi. On compiling queries in first-order databases. *Journal of the ACM*, 31(1):47–85, 1984.

[95] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[96] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report 90/?, Glasgow University, Glasgow G12 8QQ, Scotland, April 1990.

[97] R. J. M. Hughes. Analysing strictness by abstract interpretation of continuations. In *Abstract Interpretation of Declarative Languages*, pages 63–102. Ellis Horwood, Chichester, England, 1987.

[98] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):865–886, August 1986.

[99] Richard Hull and Jianwen Su. On the expressive power of database queries with intermediate types. *Journal of Computer and System Sciences*, 43:219–267, 1991.

[100] Richard Hull and Chee K. Yap. The Format model: A theory of database organisation. *Journal of the ACM*, 31(3):518–537, July 1984.

[101] Tomasz Imielinski, Shamim Naqvi, and Kumar Vadaparty. Incomplete objects — a data model for design and planning applications. In James Clifford and Roger King, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data, Denver, Colorado, May 1991*, pages 288–297. ACM Press, 1991.

[102] Tomasz Imielinski, Shamim Naqvi, and Kumar Vadaparty. Querying design and planning databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *LNCS 566: Deductive and Object Oriented Databases*, pages 524–545, Berlin, 1991. Springer-Verlag.

[103] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[104] Neil Immerman, Sushant Patnaik, and David Stemple. The expressiveness of a family of finite set languages. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, pages 37–52, 1991.

[105] ISO. *Standard 8824. Information Processing Systems. Open Systems Interconnection. Specification of Abstraction Syntax Notation One (ASN.1)*, 1987.

[106] ISO. *Standard 9075. Information Processing Systems. Database Language SQL*, 1987.

[107] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[108] G. Jaeschke and H. J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proceedings ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, California, March 1982.

[109] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[110] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[111] D. Johnson. *A Catalog of Complexity Classes*, Chapter 2, pages 67–161. North Holland, 1990.

[112] P. H. J. Kelly. *Functional Languages for Loosely-Coupled Multiprocessors*. PhD thesis, Department of Computing, Imperial College of Science and Technology, South Kensington, London SW7 2BZ, England, 1987.

[113] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[114] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[115] Won Kim. A new way to compute the product and join of relations. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 179–187, 1980.

[116] Won Kim. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA, 1990.

[117] Aviel Klausner and Nathan Goodman. Multirelations: Semantics and languages. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of 11th International Conference on Very Large Databases, Stockholm, August 1985*, pages 251–258, Los Altos, CA, August 1985. Morgan Kaufmann.

[118] H. Kleisli. Every standard construction is induced by a pair of adjoint functors. *Proc. Amer. Math. Soc.*, 16:544–546, 1965.

[119] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[120] P. Kolaitis and C. Papadimitriou. Why not negation by fixedpoint? In *Proceedings of 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1988.

[121] J. B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory Series A*, 13:297–305, 1972.

[122] G. M. Kuper. Logic programming with sets. In *Proceedings of 6th ACM Symposium on Principles of Database Systems*, pages 11–20, 1987.

[123] G. M. Kuper. On the expressive power of logic programming languages with sets. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 10–14, Los Angeles, California, 1988.

[124] Gabriel M. Kuper and Moshe Y. Vardi. On the complexity of queries in the logical data model. *Theoretical Computer Science*, 116(1):33–58, August 1993.

[125] K. Kupert, G. Saake, and L. Wegner. Duplicate detection and deletion in the extended NF^2 data model. In W. Litwin and H. J. Schek, editors, *LNCS 367: Foundation of Data Organization and Algorithms*, pages 83–101. Springer-Verlag, June 1989.

[126] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[127] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, Volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, London, 1986.

[128] Leonid Libkin. An elementary proof that upper and lower powerdomain constructions commute. *Bulletin of the EATCS*, 48:175–177, 1992.

[129] Leonid Libkin. *Aspects of Partial Information in Databases*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. In preparation.

[130] Leonid Libkin and Limsoon Wong. Query languages for bags. Technical Report MS-CIS-93-36/L&C 59, University of Pennsylvania, Philadelphia, PA 19104, March 1993.

[131] Leonid Libkin and Limsoon Wong. Semantic representations and query languages for orsets. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 37–48, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-88.

[132] Leonid Libkin and Limsoon Wong. Aggregate functions, conservative extension, and linear orders. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 282–294. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[133] Leonid Libkin and Limsoon Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49(6):273–280, March 1994. See also UPenn Technical Report MS-CIS-93-60.

[134] Leonid Libkin and Limsoon Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 155–166, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-93-95.

[135] Leonid Libkin and Limsoon Wong. Some properties of query languages for bags. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 97–114. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-36.

[136] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2:93–109, 1985.

[137] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, Berlin, 1971.

[138] I. A. Macleod. A database management system for document retrieval applications. *Information Systems*, 6(2), 1981.

[139] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.

[140] David Maier and Bennet Vance. A call to order. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 1–16, Washington, D. C., May 1993.

[141] Akifumi Makinouchi. A consideration on normal form of not necessarily normalised relation in the relational data model. In *Proceedings of 3rd International Conference on Very Large Databases, Tokyo, Japan*, pages 447–453, October 1977.

[142] Ernest G. Manes. *Algebraic Theories*, Volume 26 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1976.

[143] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[144] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[145] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[146] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[147] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of 16th International Conference on Very Large Databases, Brisbane, Australia, August 1990*, pages 264–277, Palo Alto, CA, August 1990. Morgan Kaufmann.

[148] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of Conference on Very Large Databases*, pages 468–478, 1988.

[149] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[150] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD. *ENTREZ: Sequences Users' Guide*, 1992. Release 1.0.

[151] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD. *NCBI ASN.1 Specification*, 1992. Revision 2.0.

[152] R. S. Nikhil. The parallel programming language Id and its compilation for parallel machines. In *Proceedings of Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1990.

[153] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[154] Atsushi Ohori. *A Study of Semantics, Types, and Languages for Databases and Object Oriented Programming*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 1989.

[155] Atsushi Ohori. Representing object identity in a pure functional language. In *Proceedings of 3rd International Conference on Database Theory, Paris, France, December 1990*, pages 41–55. Springer-Verlag, December 1990.

[156] S. L. Osborn. Identity, equality, and query optimization. In K. R. Dittrich, editor, *LNCS 334: Advances in Object Oriented Databases Systems*. Springer-Verlag, 1988.

[157] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[158] Jan Paredaens, February 1993. Private communication at Bellcore Collection Type Workshop.

[159] Jan Paredaens and Dirk Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[160] P. Pearson, N. Matheson, N Flescher, and R. J. Robbins. The GDB human genome data base anno 1992. *Nucleic Acids Research*, 20:2201–2206, 1992.

[161] P.L. Pearson. The genome data base (GDB), a human genome mapping repository. *Nucleic Acids Research*, 19:2237–2239, 1991.

[162] M. Petre and R. L. Winder. Issues governing the suitability of programming languages for programming tasks. In *People and Computers IV: Proceedings of HCI'88*, Cambridge, 1988. Cambridge University Press.

[163] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible rule-based query rewrite optimization in Starburst. *SIGMOD Record*, 21(2):39–48, June 1992.

[164] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proceedings of International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[165] Didier Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[166] Didier Remy. Efficient representation of extensible records. In P. Lee, editor, *Proceedings of ACM SIGPLAN Workshop on ML and its Applications*, pages 12–16, 1992.

[167] John Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *LNCS 94: Proceedings of Aarhus Workshop on Semantics-Directed Compiler Generation*. Springer-Verlag, January 1980.

[168] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):390–417, December 1988.

[169] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[170] David A. Schmidt. *Denotational Semantics: A Methodology For Language Development.* Allyn and Bacon, Boston, 1986.

[171] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979. Reprinted in *Readings in Database Systems*, Morgan-Kaufmann, 1988.

[172] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of 6th ACM Conference on Functional Programming and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[173] Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Transaction on Database Systems*, 14(3):322–368, September 1989.

[174] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[175] David Stemple and Tim Sheard. A recursive base for database programming. In J. W. Schmidt and A. A. Stogny, editors, *LNCS 504: Next Generation Information System Technology*, pages 311–332, Berlin, 1990. Springer-Verlag.

[176] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[177] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, July 1987.

[178] Dan Suciu. Fixpoints and bounded fixpoints for complex objects. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 263–281. Springer-Verlag, January 1994. See also UPenn Technical Report MS-CIS-93-32.

[179] Dan Suciu and Jan Paredaens. Any algorithm in the complex object algebra needs exponential space to compute transitive closure. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 201–209, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-94-04.

[180] Dan Suciu and Limsoon Wong. On two forms of structural recursion, July 1993. Manuscript available from `limsoon@saul.cis.upenn.edu`.

[181] Abdullah U. Tansel and Lucy Garnett. On Roth, Korth, and Silberschatz's extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 17(2):374–383, June 1992.

[182] Y. Tateno, Y. Ugawa, Y. Yamazaki, H. Hayashida, N. Saitou, and T. Gojobori. The DNA data bank of Japan. In W. C. Barker, editor, *CODATA Bulletin: Information Integration for Biological Macromolecules*, pages 74–75. 1991.

[183] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis and F. P. Preparata, editors, *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

[184] P. W. Trinder. *A Functional Database.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, England, December 1989.

[185] P. W. Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, pages 49–62. Morgan Kaufmann, August 1991.

[186] P. W. Trinder and P. L. Wadler. Improving list comprehension database queries. In *Proceedings of TENCON'89, Bombay, India*, pages 186–192, November 1989.

[187] David Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and David Turner, editors, *Functional Programming and its Applications.* Cambridge University Press, 1982.

[188] David Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *LNCS 201: Proceedings of Conference on Functional Programming Languages and Computer Architecture, Nancy, 1985*, pages 1–16. Springer-Verlag, 1985.

[189] Jeffrey D. Ullman. *Principle of Database Systems*. Pitman, 2nd edition, 1982.

[190] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies*. Computer Science Press, Rockvill, MD 20850, 1989.

[191] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[192] A. van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1992.

[193] L. Vielle. From QSQ towards QoSaQ: Global optimization of recursive queries. In *Proceedings of 2nd International Conference on Expert Database Systems*, 1988.

[194] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 45–52, Austin, Texas, August 1984.

[195] Philip Wadler. Listlessness is better than laziness II. In H. Ganzinger and N. D. Jones, editors, *LNCS 217: Programs as Data Objects*. Springer-Verlag, October 1985.

[196] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of Conference on Principles of Programming Languages*, pages 307–313, 1987.

[197] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[198] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[199] David A. Watt and Phil Trinder. Towards a theory of bulk types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

[200] W. Wechler. *Universal Algebra for Computer Scientists*, Volume 25 of *EATCS Monograph on Theoretical Computer Science*. Springer-Verlag, Berlin, 1992.

[201] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[202] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, second edition, February 1984.

[203] Limsoon Wong. A conservative property of a nested relational query language. Technical Report MS-CIS-92-59/L&C 48, University of Pennsylvania, Philadelphia, Pennsylvania 19104, July 1992.

[204] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-59.